
Real world deployments of AI-assisted tools for compilation error repair and program retrieval

A Thesis Submitted

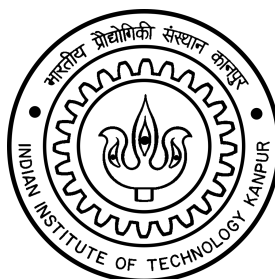
In Partial Fulfillment of the Requirements

For the Degree of Master of Technology

by

Sharath HP

19111082



to the

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur

June 2021

Declaration

This is to certify that the thesis titled “**REAL WORLD DEPLOYMENTS OF AI-ASSISTED TOOLS FOR COMPILATION ERROR REPAIR AND PROGRAM RETRIEVAL**” has been authored by me. It presents the research conducted by me under the supervision of **PURUSHOTTAM KAR, AMEY KARKARE**.

To the best of my knowledge, it is an original work, both in terms of research content and narrative, and has not been submitted elsewhere, in part or in full, for a degree. Further, due credit has been attributed to the relevant state-of-the-art and collaborations (if any) with appropriate citations and acknowledgments, in line with established norms and practices.



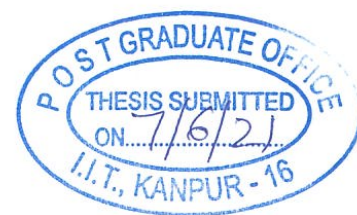
Name: Sharath HP (19111082)

Program: Master of Technology

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur, Kanpur, 208016.

June 2021



Certificate

It is certified that the work contained in the thesis titled “**REAL WORLD DEPLOYMENTS OF AI-ASSISTED TOOLS FOR COMPILATION ERROR REPAIR AND PROGRAM RETRIEVAL**” by **SHARATH HP** has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Two handwritten signatures in black ink. The first signature is "PKR" with a stylized flourish underneath. The second signature is "Amey Karkare" written in a cursive style.

Prof. Purushottam Kar, Prof. Amey Karkare
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
Kanpur, 208016.
June 2021

Abstract

Name of student: **Sharath HP** Roll no: **19111082**

Degree for which submitted: **Master of Technology**

Department: **Department of Computer Science and Engineering**

Thesis title: **Real world deployments of AI-assisted tools for compilation error repair and program retrieval**

Name of thesis supervisor: **Purushottam Kar, Amey Karkare**

Month and year of thesis submission: **June 2021**

Use of technology, specifically AI, in a pedagogical setting has seen a tremendous growth in the recent years. A good amount of research has gone into proposing such AI-assisted systems, those that are useful when applied in an academic setting. While conceptualizing such tools is important, equally important is their development and deployment in a live environment. This thesis presents the design and development of AI-assisted systems for compilation error repair and program retrieval. Specifically, this thesis outlines the development of PRIORITY, a web application and PYEDU, a VS Code extension, both targeted mainly towards pedagogical applications. PRIORITY complements the work reported in the companion thesis "Advancements in AI-assisted compilation error repair and program retrieval" submitted alongside the present one and enables tutors of the ESC101 course, an introductory programming course offered to freshman students at the Indian Institute of Technology Kanpur, get access to a search-able database of programming assignments from previous offerings of the course. PYEDU comprises the development a VS Code extension and a back-end server which is augmented by an AI system to predict fixes for programming errors. The VS Code extension, mainly targeted towards freshman CS1 students, provides an interface to view helpful feedback on programming errors. In a nutshell, this thesis makes contributions in the following two ways:

PRIORITY The thesis develops a web portal, that provides users access to a search-able inventory of computer programs, in the C programming language. The inventory is a collection of questions prepared by the tutors in past offerings of the course, that are exposed with the help of an AI system described in the companion thesis listed above. The UI keeps in mind simplicity and ease of use at its core. It further describes the design and implementation of a back-end that integrates with the above.

PYEDU The thesis also develops a VS Code extension that helps students fix programming errors in the Python programming language. This is made possible by presenting helpful feedback on syntax errors. The feedback is designed to be user-friendly, intuitive and relevant to the users. It further develops a back-end server which is augmented with an AI system in the form of Py-MACER, a Python adaptation of an enhanced version of MACER, a novel program repair tool for which details are available in the companion thesis listed above.

Acknowledgments

I take immense pleasure in expressing words of gratitude for all the people who have played a part in moulding me and have accompanied me in the past two years of my journey at the Indian Institute of Technology Kanpur.

I express my heartfelt gratitude to Purushottam Kar, Amey Karkare, who have been a constant source of support and motivation in the course of completing this thesis. They were always available for me, to clarify different aspects of my work and give a broad vision of the end goal. They have been very encouraging, extremely supportive and one of the best people I have interacted with, as teachers, as guides and bear an amazing personality.

I sincerely thank Fahad Shaikh with whom I have had constant discussions and knowledge sharing sessions in the course of completing my thesis work. A special thanks to Umair Z. Ahmed, post-doc researcher at National University of Singapore, who helped with some parts of my thesis work.

I extend my appreciation and thanks to the members of PROSE Team from Microsoft Research Lab at Redmond. I would especially like to thank Margaret Price for sponsoring the project on building the VS Code extension, Sumit Gulwani for leading the initiative, Titus Barik, Danny Simmons, Ivan Radicek and Gustavo Soares among others for providing active feedback during development of the extension.

Although the past year has been rather depressing in terms of lack of peer association on campus, I thank all my batch-mates, who have been an essential part of my academic and personal journey, the past two years. I extend a special thanks to my wife Lakshmi Priya, an architect by profession, who gave me valuable feedback, on designing some of the diagrams included in this thesis. I also extend special thanks to all the others who encouraged me, academically and non-technically alike, including my parents, friends and well-wishers.

This thesis was compiled using a template graciously made available by Olivier Commowick http://olivier.commowick.org/thesis_template.php. The template was suitably modified to adapt to the requirements of the Indian Institute of Technology Kanpur.

Sharath HP
June 2021

Contents

Acknowledgments	ix
Contents	xi
List of Figures	xiii
List of Tables	xv
I PYEDU	1
1 Introduction	3
1.1 Background	4
1.2 Our Contributions	5
2 PyMACER	7
2.1 Introduction	7
2.2 Overview	9
2.3 PyMACER Technique	10
2.4 Experiments and Evaluation	14
3 VS Code Extension Client	19
3.1 Goals of PYEDU	19
3.2 Architectural Overview	21
3.3 Extension Contributions	23
3.4 Feature Modules	26
3.5 Application Flow	31
4 Back-End Server	35
4.1 Overview	35
4.2 API Description	36
4.3 Illustration	38
4.4 Unit Tests	38
5 Conclusion and Future Work	41

II PRIORITY	43
6 Introduction	45
6.1 Background	46
6.2 Our Contributions	46
7 Web-Application Portal	49
7.1 Goals of PRIORITY portal	50
7.2 Architectural Overview	51
7.3 Feature Modules	53
7.4 Application Flow	63
8 Back-End Server	65
8.1 Database Description	65
8.2 System Overview	70
8.3 API Description	74
9 Deployment and User Feedback	79
9.1 Deployment	79
9.2 User Feedback	80
10 Conclusion and Future Work	85
Bibliography	87

List of Figures

2.1	Error Distribution on data-set 1	15
2.2	Error Distribution on data-set 2	15
2.3	Error Distribution on the combined data-set	15
3.1	Architectural Overview of PYEDU frontend	23
3.2	Menu Icons of the VS Code Extension	25
3.3	Code Actions Provider	29
3.4	Decorations Provider	31
3.5	Flow of PYEDU Application	33
7.1	Angular Application Architecture	52
7.2	Architectural Overview of PRIORITY Frontend	53
7.3	Authentication Module	54
7.4	Login Component	55
7.5	Login Component View	55
7.6	Display Module	56
7.7	Problem Search Component	57
7.8	Problem Search Component View	57
7.9	Problem Display Component	58
7.10	Problem Display Split Component View	59
7.11	Feedback Module	60
7.12	Rating Component View	62
7.13	Label Selection Component	62
7.14	Label Selection Component View	63
7.15	PRIORITY Portal Application Flow	63
8.1	Entity Relationship Diagram	67
8.2	Django App Architecture	71
8.3	Architectural Overview of PRIORITY Backend	71
9.1	PRIORITY Deployment Architecture	80
9.2	Plot of # of tutors (vertical) against range of # questions created (horizontal)	81
9.3	Fraction of questions set with assistance from PRIORITY (Phase I)	82
9.4	Average number of queries made before getting a ‘satisfactory’ result	82
9.5	Average ranked position of a ‘satisfactory’ result	83
9.6	Reduction in time and effort to frame questions with help of PRIORITY as compared to without using PRIORITY	83
9.7	Estimation on reduction in time and effort to frame questions with the help of tools like PRIORITY	83

List of Tables

2.1	DS1	16
2.2	Merged DS1 and DS2	16
2.3	Train on DS1, test on DS2	16
2.4	Train on DS2, test on DS1	16
3.1	Extension Key Bindings	26
3.2	Extension Configuration Properties	26
8.1	Click Table	67
8.2	User Table	67
8.3	Feedback Table	68
8.4	Problems Table	69
8.5	Text Feedback Table	69
8.6	Feedback Rating Table	70

Part I

PYEDU

Introduction

Contents

1.1 Background	4
1.2 Our Contributions	5

One of the most intimidating experiences of a student beginning to learn programming is fixing compilation errors [7]. There is a study [11] that indicates that the errors novice programmers make, when undertaking such a course, follows a particular distribution and frequency pattern. And it is found that it follows a heavy tail distribution, with most of the errors falling in a rather small bucket of error types. This provides immense opportunity to find out if we could work on these errors, to suggest automated fixes albeit in a helpful and intuitive way. In fact, over the past few years, there has been some exploration in enabling intelligent tutoring systems in order to assist smooth conduction of introductory programming courses taught in educational institutions worldwide [4]. One of the ways to help fix these errors is to provide a helpful feedback, that is relevant and intuitive to the students, at least on the major and most common of the error types. This is especially true of MOOCs (Massive Open Online Courses), where instructor(s)/ tutor(s) interacting with all of the students to meet their learning objectives is a costly demand. Some of the common observations that inspire the work proposed in this thesis are as follows:

1. Compiler error messages are often cryptic [13] and not helpful enough for novice programmers to resolve the errors by themselves. The compiler designers think of messages as ways of telling what went wrong, and not report what to do to make things right, being unaware of user intent and the multitude of possibilities that exist therein.
2. Even if the error messages were made somewhat better, the ones that suit novice programmers might not suit experts and vice versa. Thus it becomes an intractable problem.
3. While some of the existing techniques for providing feedback to students on fixing compiler errors only target specific category of error types, many other generic tools that try to achieve the objective of providing useful feedback to the students, are often simply an extension of the compiler reported error message. Thus it is, at best, only as helpful (or not) as the compiler reported error messages.

In order to overcome some of these limitations, we propose enabling systems that provide AI-assisted feedback, relevant to the student needs and also present them in an intuitive, user friendly way customized to a particular audience.

1.1 Background

Many of the works in literature that try to achieve the objective of helping CS1 students fix erroneous programs, can broadly be classified into three categories viz., the ones that actually predict a correct version of the broken program, the ones that work on enhancing the compiler generated error messages, and the ones that attempt to provide some sort of generated feedback to help the students. We explore some of these below.

Compiler Error Repair There have been advancements in compilation error repair, especially syntactic errors, in the recent past, we refer to two state-of-the-art approaches in this direction. Like most other approaches, DrRepair [14], which proposes a Graph Attention based LSTM network for Compiler Error correction using compiler diagnostic feedback, is based on a deep learning framework. It takes as input the source program along with the compiler diagnostic feedback, encodes them via LSTM and graph attention layers, and decodes the error line and the corresponding repaired code. MACER [5], unlike most other techniques, uses light weight ML techniques and incorporates a modular pipeline to predict fixes to errors. It identifies the error line location (lines that need repair), identifies the type of repair needed on each line (repair-class of the line) and also points to where in the particular line the repair has to be applied (repair-profile of the line). Because of the modular nature, MACER could also target specific error types.

Enhanced Error Messages Most techniques that report enhanced error messages instead of the compiler error messages are static in their approach, while some do generate them based on historical user submissions. There is some debate in the research community as to what is the optimal way to present error messages to students, where some argue that additional information doesn't help students [9] significantly [10], while some others are in favour of the claim [3]. While this is true, it is also a case that the same error messages might not help cover all user demographics, for instance consider novice programmers v/s experienced developers.

Example based Feedback While many of the existing works in literature focus on compilation error repair and predicting the correct program without giving much regards to the fix desired by the student, in a way that could help them learn programming concepts, some works target the specific area of providing a more informative custom feedback to the students to fix the errors by themselves. TEGCER [2], HelpMeOut [8], are some of the state-of-the-art proposals in this regard. These approaches are based on suggesting relevant programming examples to demonstrate to the students on how to correct the erroneous programs at their end. TEGCER also showed that using this technique without the intervention of a manual tutor helped the students resolve the errors 25% faster on average. This was achieved by integrating the tool with the Prutor IDE and offering it as part of ESC101 course in a semester of freshman undergrads.

In addition to TEGCER described above, there are some other existing applications that have demonstrably been used in an actual study setting. Help50 is one such tool that is developed at Harvard University, offered in the CS50 IDE, as part of a family of CS50 Tools. Help50 helps print less arcane compiler and Valgrind messages. Although helpful, the power of this tool is limited to the extent to which compiler error messages are themselves helpful (which may sometimes also be

misleading). Alongside this, there have been some developments in the open source community, in the form of VSCODE extensions, one of the popular open source IDEs available right now. These include linting tools like Pylint and others static code analysis tools like Pylance. While these tools are powerful in what they do, they do not specifically target the problem of resolving syntax errors in student programs and lack any AI assistance in the back-end.

While making a choice for the AI-assisted system to augment the back-end, we chose to focus on how lightweight it is, in terms of prediction, on training, and also how amenable it is for the purpose of its adoption/ porting to popular programming languages other than the ones where the original technique was applied to (which in most cases is C/ C++). Another matter of choice for the tool is the deployment environment (or IDE). Most of the techniques described above target the C/C++ programming language. Python, a mainstream programming language, is gaining traction as the medium of instruction in CS1 courses across educational institutions in the recent times. In addition to supporting this, the IDE must also be used widely and popular among the student community and the developer community alike.

1.2 Our Contributions

As part of this thesis, we design and develop PYEDU, a user friendly tool for helping novice programmers fix programming errors. The tool is developed based on a client-server model. It consists of two main components, the front-end (or user interface) and the back-end (or hosted server). What follows is a brief note on the design and implementation aspects of PYEDU.

Front-end design and development User friendliness and an intuitive presentation of the contents are at the core of the design principles for the tool front-end. The user interface is built to be visually appealing to the users of the tool and the display hints are presented intuitively. The deployment environment (VS Code) provides all the necessary APIs to facilitate such an implementation.

Back-end design and development There are two main components in the back-end viz., the server that handles all the communication and interfaces with the client(s) and the AI-assisted system that predicts the error fix. An extension of the MACER tool-chain [12], solves the problem of suggesting relevant fixes on programming errors, but its current implementation addresses the C programming language only. In the recent times, however, more and more institutes have started to incorporate Python programming language as the medium of instruction in such courses. Thus there is an immediate need for such a system which works with Python. One of the main tasks here is to port the previous technique to fix errors in Python programming, in order to make it widely usable. While the pipeline remains the same by and large, it required significant non-trivial changes to some of its modules. While this AI-assisted system is an augmentation to the server at the back-end, the current version of the server itself is an instance of a flask application.

Deployment Environment We choose VS Code as the platform for deploying our tool, given that it is a widely popular open source IDE and is popular in the student community and the de-

veloper community alike. Besides, its integration with GitHub, in the purported release of GitHub Codespaces¹, means that going forward it is only going to be adopted by even more people, promoting its large scale adoption in the developer community as well. The potential impact it has in its outreach, targeting students and developers with experience, makes it an ideal candidate for deploying such a tool. The end result is PYEDU, due to being shipped with a version of VS Code in the near future term. This tool is purported to be used in a programming course offered to freshman students of a reputed educational institution, in the upcoming academic year.

The following chapters provide a detailed explanation of the above mentioned contributions to PYEDU. Chapter 2 describes in detail the python adaptation of MACER's extension [12]. Chapter 3 provides details on the key components of the user interface exposed by the VS Code extension, its design and development. The server at the back end, and development are detailed in chapter 4.

¹<https://github.com/features/codespaces>

PyMACER

Contents

2.1	Introduction	7
2.1.1	Related Work	8
2.1.2	Python Dataset	8
2.2	Overview	9
2.2.1	MACER	9
2.2.2	MACER++	10
2.3	PyMACER Technique	10
2.3.1	Pre-processing the Data	10
2.3.2	C vs Python	11
2.3.3	Adapting MACER++ for Python	11
2.3.3.1	Abstraction of source programs	11
2.3.3.2	Uninformative Compiler diagnostic	12
2.3.3.3	Incremental repair	13
2.3.3.4	Concretization Step	13
2.3.3.5	Fixing Indentation Errors	13
2.4	Experiments and Evaluation	14
2.4.1	Data-sets	14
2.4.2	Error Distribution	14
2.4.3	Results	15

Abstract *This chapter is dedicated to describing the development of the AI sub-system on the server side (back-end), which is used to predict the fixes to buggy user programs. We begin by giving an overview of some techniques for compilation error repair. This is followed by a brief overview of the MACER technique, an end-to-end system for suggesting repairs to programs with compiler errors. We then describe the efforts in porting MACER++ to python, highlighting the challenges involved. The chapter concludes by discussing some results obtained while experimenting with the python datasets used.*

2.1 Introduction

As discussed previously, there are several techniques for automated compilation error repair. After exploring a few of the approaches, two state-of-the-art approaches, which are of specific interest to us, are described below.

2.1.1 Related Work

DrRepair [14] uses graph attention based LSTM network to achieve this. They take the source program and the compiler diagnostic message, encode them through bi-directional LSTM networks and a graph attention layer, to finally output the error line and the repaired code. We also refer to MACER++ [12], an enhanced version of MACER [5]. MACER proposes a light-weight, modular technique for automated error correction. They set up a repair pipeline that is mainly segregated into two phases - error line localization and repair application. The repair application is further segregated into finding what type of fix is needed on the repair line (referred to as repair class) and where in the repair line the fix has to be applied (referred to as repair profile).

DrRepair works on the C/C++ programming language, whereas the MACER technique works on the C programming language originally. Both would need efforts with respect to porting them to more popular programming languages. We would also anticipate some effort to amend the technique albeit make some modifications to them, in order to make them work with Python as such. DrRepair, while it does perform well, uses techniques that are costly to train nevertheless. On the flip side, MACER++, which performs decently in comparison, is a very light-weight in its approach, and demonstrably takes as less as a few minutes to train as against a few days to train DrRepair. Besides this, the modular nature that it brings to the fore (and the notion of repair classes) is especially useful for working with and providing users meaningful diagnostic feedback on compilation errors.

2.1.2 Python Dataset

Given the rising popularity of the Python programming language, especially in the student community, and its mainstream adoption in schools and colleges, it was deemed necessary to port the chosen technique to work with Python. As such as it has been the popular choice of language for instructors teaching introductory courses on programming. In this context, it is also important to note that while the previously mentioned techniques work well after being trained on the original data sets (which span in the range of 1000s of training data points), the training data available for the Python programming language is rather limited. The dataset that we currently have is a dump of python programs that were obtained from two offerings of a Python Programming course taught on the Prutor IDE [6]. Prutor saves snapshots of student submissions (including both buggy and correct submissions) at regular intervals. After some pre-processing of this data dump, we were able to generate an aligned dataset (source-target programs differing in a single line). The subsequent sections of the chapter elaborates on this. This would mean that the technique chosen must also be able to work with a rather scanty data to start with.

In light of the above discussion, after evaluating the two approaches for automated compiler error repair mentioned above, MACER++ was chosen as the go-to technique owing to the various benefits it brings to the table. Most importantly, given the flexibility and the modularity of its pipeline and the fact that the amount of data available to us right now is rather less (which very much acts against favour of DrRepair), it was a favorable choice for the AI system in the back-end.

2.2 Overview

Before we delve into explaining the MACER [5] technique, we shall take a look at why at all such a technique may be required for assisting students fix compilation errors in the first place. The below points give us an insight into this:

- Compiler designers design error messages to indicate what went wrong. Users want to know what has to be done to make things right.
- User intent is hard to guess. For instance, an expression like $i = 0$ in a conditional statement, may either be intentional or a mistake on the part of the user, where it must have been $i == 0$. Anticipating all such possibilities in an error message is infeasible for compiler designers.
- Error messages may need to be different for different demographics. The compiler diagnostic appropriate for a beginner would be different from that for an experienced programmer.

2.2.1 MACER

After recognizing the need for such a technique, we provide some details about the specifics of the enhanced MACER technique, which we would be adopting for use in our tool. Being an extension of the MACER technique, the entire pipeline is more or less the same. We provide a brief explanation of the MACER pipeline below. MACER segregates the entire process of repairing compiler errors into the following steps:

1. **Repair line localization:** Locate the lines in the source program that require repair. This uses a simple technique, enumerating all compiler reported error line numbers and a line just above/ below them, as explained in TRACER [1].
2. **Featurization:** The prospective repair line is then converted to its abstracted form (replaced with compiler provided abstract tokens) and the input is encoded into a bag-of-words feature representation.
3. **Repair Class Prediction:** An ML model then predicts the repair class (consisting of tokens to be inserted/ deleted to perform repair) using the input feature vector.
4. **Repair Localization:** Predicts bigrams within the source line where repair has to be applied.
5. **Repair Application:** Predicted repairs are applied at the locations predicted above.
6. **Repair Concretization:** Replace back the abstract tokens with proper identifiers with the help of symbol table.

The entire repair process, including generation of the repair classes, is automated. The modular nature of MACER is its biggest advantage, something that we could exploit. While fixing compilation errors, this would enable us to extract appropriate information, with varying levels of details relevant to the appropriate fix to be made.

Some of the possibilities and potential ways to generate helpful feedback, the celebration behind design choices for the presentation of visual feedback as part of the VS Code extension front-end, are further elaborated in Chapter 3.

2.2.2 MACER++

Recent advancements to AI compilation error repair [12] propose MACER++, an enhanced version of the MACER technique described earlier. These enhancements improve some of the modules in MACER, while keeping the entire pipeline more or less the same. Hence, the entire subject matter discussed in the previous section still holds true. Using MACER++, we get all the benefits of using MACER, only that it's more efficient now.

In addition to these it also introduces a novel technique to generate synthetic data. The synthetic data generation technique uses existing data (source programs) and generates new samples with realistic errors, ones that resemble errors made by novice programmers. This is also very useful, considering the paucity of the Python dataset that we have to deal with.

2.3 PyMACER Technique

Given its rising popularity and adoption, we are mainly interested in developing a tool that suggests interactive and helpful feedback to students on errors in Python programs. As explained earlier, the MACER++ pipeline was originally implemented for the C programming language. We implement PyMACER, an adaptation of MACER++ to fix errors in the Python programming language, for incorporating it as backend AI system.

2.3.1 Pre-processing the Data

While MACER++ was trained on an aligned dataset, of around 17k programs (TRACER Single dataset [1]), we do not have the same privilege with respect to Python. The dataset that we have is rather limited as described in section 2.1. The original data dump from the Prutor IDE needs to be subject to some pre-processing to get the (aligned) source-target pairs of programs (which differ in a single line of source). In this process, MACER++ uses the LLVM compiler framework to obtain the abstract representation of the source program. Python, being a dynamically typed language, the abstract tokens obtained at the lexical analyser (using ANTLR) isn't much informative. For instance all functions, classes and identifiers are classified under a single token type (NAME). We use the AST to get more type information. There are some subtle nuances to this, which are described in section 2.3.3. With this, we are able to get basic type inferences so that the abstract representations of the programs are meaningful for our technique to work with.

Finally, after pre-processing the data, we were able to generate a combined dataset that resulted in just about 2k source programs. Similar to the TRACER Single dataset, it contains pairs of source programs and target programs differing in just a single source line, and contains other useful meta data. Going forward we shall refer to this dataset as the Python dataset.

Any machine learning technique requires a decent amount of data to train the model. This is especially true of deep learning techniques. PyMACER will have to work with a rather scarce data in Python. There is an effort to augment this dataset with synthetically generated data, using the technique from MACER++, but overall it is still pretty limited.

2.3.2 C vs Python

There are some definite differences between Python language, which is Interpreted, and a compiled programming language as C. The native Python interpreter can't be used for generating diagnostic information, as that would introduce additional dependency in terms of test case requirements (as otherwise we might not be able to run the program entirely). However the source program can be compiled to bytecode representation, thereby capturing all compile time errors, with the help of some tools, like the `py_compile` module for instance. The following points highlight some of the other differences seen:

- Unlike C, not all compile time errors are available at once in Python. The native `py_compile` module reports only one compilation error at a time.
- Python is a dynamically typed programming language, whereas C is statically typed. This might be significant, as we suspect that types are a rich source of information for the machine learning model to learn from. While some type inferences could be done on the Python programs, it is not straight-forward to do so. Besides with a limited dataset, the use of type information may not actually benefit us.
- The category of errors reported by the compiler/interpreter respectively, differ significantly across the two programming languages. In Python, there is a broad category of errors (constitutes about 60% of the errors seen in the Python dataset), all tagged under a common label "SyntaxError: invalid syntax", which doesn't really help the student. This is especially seen by the poor diagnostic feedback in such cases, which are really not helpful to the students, making it even more necessary for such a system to be in place to assist them.
- Unlike C, a lot of errors are only caught at runtime in Python (for instance some type related errors). This poses another challenge in terms of early detection of errors, making such a tool less effective. For reasons stated earlier, we would not be able to run the python program to check for errors. This is also risky, as it imposes additional security concerns in the context of program execution.

The following section elaborates more on these challenges from an implementation perspective.

2.3.3 Adapting MACER++ for Python

The previous sections mainly focused on the Python dataset available for PyMACER to work with as well some pointers on the conspicuous differences between C and Python as programming languages, something that affects the direct adaptability of MACER++ to Python. We take a look at the challenges posed herein and the efforts made to mitigate them in the process of building the AI-system in the backend for PYEDU. The following sections elaborate on these.

2.3.3.1 Abstraction of source programs

Given that MACER++ works with an abstract representation of source programs, one of the main steps in preparation of the input feature vector is generation of the abstracted source programs. Originally, MACER++ uses the LLVM compiler framework to generate an abstract representation

for C programs. To replicate this for python programs we use ANTLR which is a parser generator and an open source Python grammar¹ based on version 3.6 of The Python Language Reference².

ANTLR Lexer Given that Python is an interpreted language, we use the Lexer generated with ANTLR for tokenizing the source program. However, the generated tokens were found to be quite uninformative, with the token abstraction resulting in a good deal of information loss. This is to be expected because of absence of static type inference in python. For instance, we observe that a token 'NAME' is used to represent all types of identifiers including modules, classes, functions etc. These types of generic tokens might effect the MACER++ pipeline negatively, and subsequently also the affect the concretization step, which is used to get the actual repair lines for the buggy source lines.

AST Parser In order to overcome the limitation of genericity of the Lexer generated tokens, we parse the source program for generating an Abstract Syntax Tree representation of the source program and then walk this tree to individually identify the different types of identifiers and thereby differentiate modules, classes, function definitions and function calls from each other. There is, however, one additional challenge this poses - generating an AST is not possible if the source program is not syntactically valid i.e., there are syntax errors in the python program. To work around this, we first truncate the program at the line number obtained from the diagnostic error message, reported by the compiler, and use this to get a prefix of the source program which is free from syntax errors. Note that this step might be repeated multiple times till we find a suitable prefix. It is also important to note that this technique could very well be used for extracting additional static type inference in the future.

We also find that novice programmers quite typically make mistakes in usage of built-in functions in python like print, input, range etc. Hence preserving these functions as is i.e., without abstracting them out could actually help in fixing these types of errors. However, we do abstract out any user-defined functions.

Another option that we experimented with, in this direction, was using the Python native tokenize module. While it proved to be useful at the start, we found that ANTLR performed much better and is more robust.

2.3.3.2 Uninformative Compiler diagnostic

We make use of the native `py_compile` module provided by Python for checking any compile time errors in the source program, which is the main and immediate target of our tool.

In C, the clang front-end of the LLVM framework was used to provide the compiler diagnostics. Compared to the clang diagnostics, the one provided by the `py_compile` module is much less informative. As explored in section 2.4, a major portion of the errors made by students fall under the "SyntaxErrors: invalid syntax" category. This is indeed a very generic error message. By this classification, we were able to identify approximately 30 different types of errors (error

¹<https://github.com/antlr/grammars-v4/tree/master/python/python3-py>

²<https://docs.python.org/3/reference/grammar.html>

classes) from the python dataset, in contrast to approximately 150 error classes identified from the C dataset.

The clang compiler, where necessary and available, also reports identifiers/ symbols that appear to be problematic, whereas Python doesn't report any such information in case of syntax errors. The usefulness of the profound diagnostic information provided by clang is justified by the improvements seen in line localization accuracies in MACER++ compared to its predecessor.

2.3.3.3 Incremental repair

In C, which is a compiled programming language, clang reports all of the errors encountered in the source program with a single compilation. As opposed to this, with Python, a single compilation reports only one syntax error (the one that is encountered first). This makes fixing the entire program a laborious process.

Thus after fixing each individual error, we would need to re-compile the program to check for additional errors. Also, with C we can determine whether a fix was successful or not just by checking the number of errors reported by the compiler, but with Python we could only work with a heuristic to determine whether the fix was successful or not. We look at the compiler reported line number to see if the previous error was fixed. If the new error is further down in the program then probably the fix was successful. It is to be noted that this is just a heuristic and may not hold true for all cases.

2.3.3.4 Concretization Step

The overall approach to Concretization did not have to change much, keeping the Concretization module more or less the same. The only change that was required for dealing with python programs was reworking with the symbol table.

2.3.3.5 Fixing Indentation Errors

As seen earlier, one of the most common types of errors that novice programmers make while programming in Python, is Indentation Error.

By and large, indentation errors generally require only an insertion or deletion of an indent (tab or spaces). Hence having to rely on AI/ ML techniques to resolve such errors might be an overkill. Moreover, the pattern and nature of such errors is highly context insensitive. So such a technique might actually perform poorly as it doesn't necessarily learn any pattern in the input source lines from the same (needs further investigation). The `py_compile` module reports Indentation Errors distinguished from Syntax Errors. This way, it makes it easier to differentiate them and target such errors and handle fixing them separately.

Subsequently, we implement a module to fix such Indentation errors viz., the Indentation Parser. Given a program with an Indentation Error, the Indentation Parser attempts to fix any Indentation issues with the source program and returns the repaired program, one with proper indentation. It is to be noted that the program may still have additional Syntax Errors that needs to be dealt with. What follows is a brief description of how the Indentation Parser works:

- The Indentation Parser moves through the program line by line keeping track of the statements which result in creation of a new block of code like an "if" statement or "else", loops, function definitions etc. until it encounters the erroneous line (reported by the compiler).
- Based on the information gathered thus far, it then determines the level of indentation required for the erroneous line and applies it.
- After the fix is applied, it then checks for more Indentation Errors and repeats the whole process. It stops when the given program compiles or when it no longer has any Indentation Error.

It is to be noted that the Indentation parser may fail in the following cases:

- If the program has an improper indent but the compiler reports a syntax error on that line (and hence not processed by it).
- Errors where the body of a certain statement block is missing i.e., we encounter constructs which requires a body, but which is absent (an error in itself). This is illustrated in the pseudo-code below

```
if condition: # missing statement block
else:
    statement block
```

2.4 Experiments and Evaluation

2.4.1 Data-sets

As mentioned earlier, the dataset that we have is extracted from the data dump of Prutor IDE, from two offerings of a beginner level Python programming course. After pre-processing the data dump, we were able to generate an aligned dataset (where source and target programs differ in a single line of source) of 1569 programs from the first data dump and that of 1105 programs from the second data dump respectively. Combining them yields a dataset (aligned as before) of 2674 source programs. Given the limited dataset, we do a customized random test train split, in a way that includes majority of the repair classes, so that the model learns nicely. With this we generate a train set of 1980 programs and a test set of 694 programs from the combined dataset. We run some ablations on PyMACER. For this we generate a separate train set of 1131 programs and a test set of 438 programs from dataset 1. Similarly, from dataset 2, we generate a train set of 845 programs and a test set of 260 programs.

2.4.2 Error Distribution

As seen earlier, we processed datasets from the two course offerings independently as well as combining them together. The following figures illustrate the frequency of errors by types in these datasets. They indicate a rough estimation of the frequency of the different types of syntax errors novice programmers make.

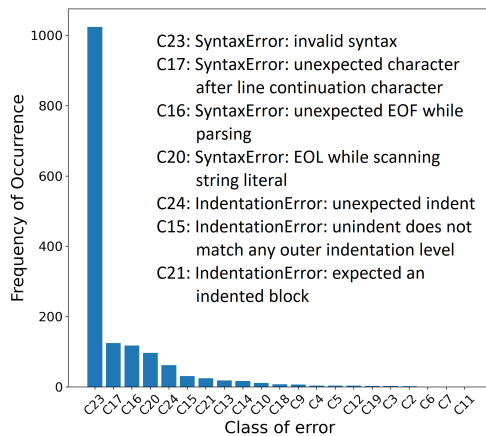


Figure 2.1: Error Distribution on data-set 1

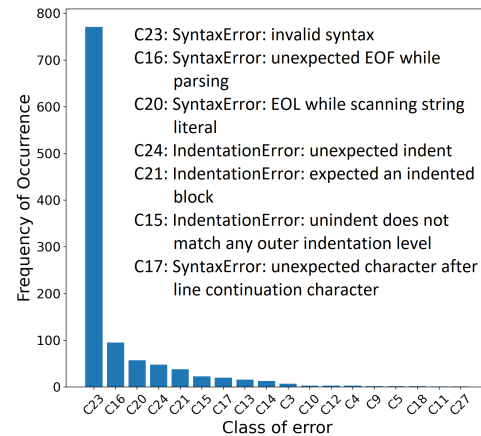


Figure 2.2: Error Distribution on data-set 2

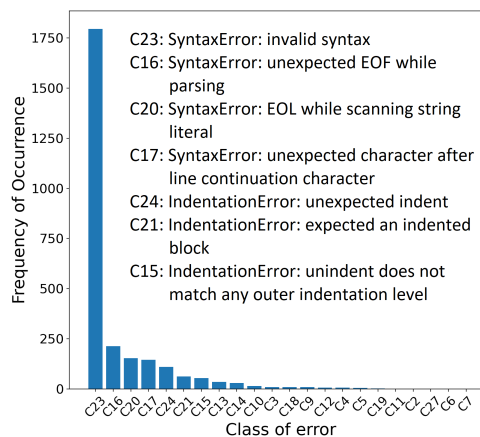


Figure 2.3: Error Distribution on the combined data-set

- From Figure 2.1, Figure 2.2 and Figure 2.3, we observe that the errors follow a heavy tail distribution and that majority of the errors fall in the first bucket, under the "SyntaxError: invalid syntax" (referred to as C23) category. The frequency of errors drop significantly beyond the 8th or 9th error type as indicated.
- Indentation Errors form another bulk portion of the errors, after the above, representative of common errors made by novice programmers.
- It is also to be noted that the pattern of syntax errors made by the students is rather common. If we look at the top 7 compile time errors by types in Figure 2.1 and 2.2, we find that they form the same set across the two datasets, with more or less a similar distribution, with the bulk of the errors still reported under the "SyntaxError: invalid syntax" category.

2.4.3 Results

We also ran an evaluation of PyMACER on both the python datasets above, independently as well as combined. We present some findings while experimenting with the two datasets next.

Notion of repair classes As explained in section 2.2, repair classes are an essential part of the MACER technique. They indicate the type of repair to be performed. Originally MACER repair classes contain three parts to it viz., the compiler reported error ID, list of tokens to be inserted and a list of tokens to be deleted. For instance, consider the repair class [C23[=[[:]] Here, C23 corresponds to the compiler reported error ID, the tokens enclosed in the first set of [], i.e., an '=' token needs to be deleted and the tokens in the second set, i.e., a ':' token needs to be inserted into the source line in identified, to make the fix.

Notion of synthetic data The generated repair classes may be used for reproducing the errors already seen in the training set. This can be applied on correct source programs to generate additional training instances for the existing, aligned, dataset [12].

Notations Please find below some handy shorthand notations used for convenient representation.

- def: Refers to the default pipeline of PyMACER (without the optimizations of MACER++)
- weid: Repair Classes without the corresponding compiler error ID information
- type: Usage of limited type inference on distinguishing identifiers as separate tokens (refer section 2.3 for details)
- synd: Usage of synthetically generated data while training
- Rep@k: Repair accuracy achieved by PyMACER by applying up to top k predicted fixes.
- DS_i: The python data-set i.

We did some ablation studies on PyMACER. In the first two experiments we looked at how much effect additional data had on PyMACER. Table 2.1 and Table 2.2 show the results corresponding to this experiment. We ran another pair of crossover experiments where training was done on one dataset, and testing on another. Table 2.3 and Table 2.4 show the results from this experiment.

Metric	Rep@1	Rep@5
def	33.2	45.3
weid	33.2	44.4
weid + type	28.1	39.1
weid + synd	34.3	47.4

Table 2.1: DS1

Metric	Rep@1	Rep@5
def	36.4	47.3
weid	38.7	48.1
weid + type	31.2	43
weid + synd	39.4	49.6

Table 2.2: Merged DS1 and DS2

Metric	Rep@1	Rep@5
def	35	45.2
weid	35.3	47.3
weid + type	28.5	39.5
weid + synd	37.5	49.5

Table 2.3: Train on DS1, test on DS2

Metric	Rep@1	Rep@5
def	35.7	45.1
weid	35.6	44.5
weid + type	26.4	38.3
weid + synd	37.2	46.9

Table 2.4: Train on DS2, test on DS1

From the previous experiments we make the following observations:

- As in case of MACER++, excluding the compiler reported error IDs helps the model perform better. This may be attributed to the fact that with collapsed repair classes and more instances for training per repair class means better accuracy.
- Using limited static type inferencing (to distinguish the variable identifiers) doesn't help the model do well, in fact, it deteriorates its performance. This may be attributed to the lack of data, which makes it favourable to have fewer tokens implying fewer repair classes.
- Using more synthetic data generated during train time is helpful. We see a significant boost in accuracy with additional data, which only highlights the need for additional training data.
- From Table 2.1 and Table 2.2 we see that increasing the data gave a significant boost to the accuracies
- As Table 2.1 and Table 2.2 show, the repair accuracies remained consistent in both, suggesting that the method generalizes well to unseen data

Overall, we found that using the combined dataset yields the best model, resulting in the best repair accuracy. We use the PyMACER model trained from this dataset for PYEDU.

VS Code Extension Client

Contents

3.1	Goals of PYEDU	19
3.2	Architectural Overview	21
3.2.1	VS Code Extensibility	21
3.2.2	PYEDU Frontend	22
3.3	Extension Contributions	23
3.3.1	Commands	23
3.3.2	Events	24
3.3.3	Menu Icons	25
3.3.4	Key-bindings	25
3.3.5	Configuration Properties	26
3.4	Feature Modules	26
3.4.1	Local Storage	26
3.4.2	Computation Module	27
3.4.3	Code Actions Provider	27
3.4.4	Decorations Provider	29
3.5	Application Flow	31

Abstract *This chapter is dedicated to describing various design and implementation aspects of the VS Code Extension, which provides the client-side interface to PYEDU, the AI-assisted tool for helping students fix programming errors. We begin by going through the main objectives set up by such a tool. This is followed by an overview of the architecture adopted. The subsequent section speaks about the main modules in the VS Code Extension and documents the purpose of each of these while describing some important aspects of their development. The chapter concludes by showcasing a flow of the application, a typical usage scenario keeping the end-users in mind.*

3.1 Goals of PYEDU

The AI-assisted tool for fixing programming errors is developed keeping novice programmers in mind, to ably assisting them in doing so. The tool, which we will refer to henceforth as PYEDU (short-name for Python Education), has three main components to it viz., the front-end user interface exposed by the VS Code extension, the back-end server and the AI-system that powers the back-end. The AI-system essentially trains a machine learning model for predicting fixes for programming errors. The training data-set is procured from a data dump of Prutor IDE of two beginner level python courses. Given a source program, the model automatically predicts the the

corrected version of the program, and it does so in a relevant manner i.e., the fixes are more or less what a student would normally expect or do themselves. More details on the AI-system can be found in chapter 2.

Programming bugs are simply a bane for novice programmers, such as freshman undergrad students, and often times becomes a great obstacle in the path of their learning. Although compilers/ interpreters may indicate that there is some error in the program, and possibly carry some location information, it often times simply isn't enough for the students to actually fix the error, mostly because of the cryptic nature of the reported diagnostics. The main objective of the tool is to help in making the process of fixing programming errors easy for such students. Eventually PYEDU would be shipped with a version of VS Code that would be used in a reputed engineering college for instructing freshman undergrad students in an Introductory Programming Course.

The target audience for PYEDU and the way the extension would benefit them is given below:

- **Students:** PYEDU is mainly targeted towards enhancing the learning experience of the students, as they start to learn programming. The tool would assist them in fixing syntax errors, by providing helpful feedback, at various levels.
- **Instructors:** PYEDU also doubles as an assistant to instructors who teach such introductory programming courses. They would be able to use the tool to possibly write custom error messages on a set of common programming mistakes students make or even just configure the tool to meet the learning objectives of students, to make their learning experience much better. This is especially true of the prevailing circumstances, where all the courses are taught online.
- **Tool Developers:** Although this may not be an immediate goal of PYEDU, but eventually as PYEDU gets adopted more and more in the academic circles, a future addition to PYEDU in terms of a logging framework that would collect user statistics, would greatly help the researchers and tool developers in enhancing the technique of automated program correction.
- **Developers:** This is a remote goal, in that as PYEDU gets used more and we are able to boost its accuracy, we may eventually target the larger audience, the world-wide developer community, who can then enable this tool for assisting them in their day-to-day development work. This may also mean that PYEDU gets adopted suitably for different programming domains.

As remarked earlier, PYEDU is set up to provide a light-weight VS Code extension in the front-end. The front-end is developed using the in-built APIs provided by VS Code. The user interface is kept simple and intuitive, and something that would give feedback at various levels to the students - based on the help requested - to meet the respective learning objectives. For instance, in case of students just wanting to see the correct version of the program, the tool may be configured to do so, but this may stunt the learning process. The exact same thing, though, could be what experienced developers may want, and hence more suitable for such audience. In another instance, a student can be shown just the location of the error i.e., the source line, and possibly the position in the source line where some fix needs to be made. In this way, the tool could thus be configured to show feedback at various levels, to meet the respective goals.

3.2 Architectural Overview

3.2.1 VS Code Extensibility

VS Code provides a rich extensibility model and there are many ways to extend its core functionality. For the sake of maintaining performance and compatibility, VS Code doesn't provide extension developers direct access to the DOM, all the extensions run in their own host process. VS Code provides a set of built-in UI components for common scenarios, such as code completion feature, which not only helps keep the experience of using it consistent across applications but also serves as a starting point for extension developers who need not re-invent the wheel for such scenarios.

Following are some of the core concepts that reason out the extensibility model adopted by VS Code:

- **Stability:** Extensions could affect startup or the overall performance and stability of VS Code, especially if the extension misbehaves. Hence the extensions are run in a separate host process. This way of isolating extension run environment ensures a responsive VS Code UI irrespective of how the extensions behave.
- **Performance:** VS Code enables lazy loading of extensions. Not all extensions may be needed for a user session, therefore they can be chosen to not be loaded at all for a particular session of using VS Code, this even helps reduce the memory footprint. This is possible by defining the activation events that VS Code provides for extensions. With this, the extensions can be activated (loaded) upon specific events such as opening of a particular file type etc.
- **Extension Manifest:** This is, perhaps, the most important file that is to be described in the process of building an extension. It contains a plethora of configuration options ranging from describing activation events, to specifying a set of contribution points that the extension can add to, to declaring module dependencies etc. Some of the important fields of concern to us are described in brief below.
 - **ID details:** These are some of the identity details such as name, version, publisher, license, description etc. that could be specified in their respective fields. These are all part of the extension manifest.
 - **contributes:** An object that describes the extension's contributions, which are, essentially a set of JSON declarations that contribute additional features to some of the existing components of VS Code. With this, extenders can contribute to a host of capabilities including commands, menus, keybindings, views etc. We refer the reader to the official documentation¹ for more details.
 - **activationEvents:** An array of the activation events defined for the extension. These are essentially a set of JSON declarations that dictate when the extension becomes activated i.e., loaded in VS Code. There are various activation events as listed on the official documentation².

¹<https://code.visualstudio.com/api/references/contribution-points>

²<https://code.visualstudio.com/api/references/activation-events>

- **dependencies**: Specifies any runtime Node.js dependencies that the extensions needs.
- **icon**: Specifies the path to an icon with a minimum resolution of 128x128 pixels.

For a more complete overview of the extension manifest we direct the reader to the official documentation³.

- **Extensibility API**: Extension Isolation enables VS Code to limit the APIs exposed by VS Code. VS Code provides a set of APIs that are useful for extension developers. This set is constantly evolving, and VS Code community actively listens to the extenders, who could request for additional APIs, based on new, discovered use cases.
- **Protocol based extensions**: This is a common pattern of extension development, where the extension is executed as an independent process and would communicate with VS Code through a protocol. Language servers are an example of this. This approach provides extenders the flexibility of choosing their own programming language for extension implementation.

For a more comprehensive overview of the VS Code extensibility model, we direct the reader to the official documentation⁴

3.2.2 PYEDU Frontend

PYEDU is developed in two parts. It is modelled on a two-tier client-server architecture. The client-side consists of just the presentation layer, which interacts with the users of the tool. The client-side component of the tool is implemented as a Visual Studio Code extension. The core functionality of the extension can be split in two.

1. Communicate with the backend server through REST API calls, to fetch hints for fixing syntax errors in the program.
2. Present the hints returned by the server to the user, intuitively and effectively, in a way so as to meet the student learning objectives.

The VS Code extension developed for PYEDU follows the same model as would any extension that is developed for VS Code. The below points speak about the different aspects that our extension contributes to the model.

- The extension is configured to be activated whenever a python file is opened in the editor window and stays put throughout the entire session of the VS Code Application.
- The extension contributes two commands - one to get diagnostic information and another to get the highlight feature capability - both pertaining to the source program in context. All of these are explained in details in section 1.2
- The extension defines keybindings for the commands noted above.
- The extension also contributes editor/ title icons to the default menu of VS Code. The icons trigger the functionality corresponding to the commands above, to which they are bound.

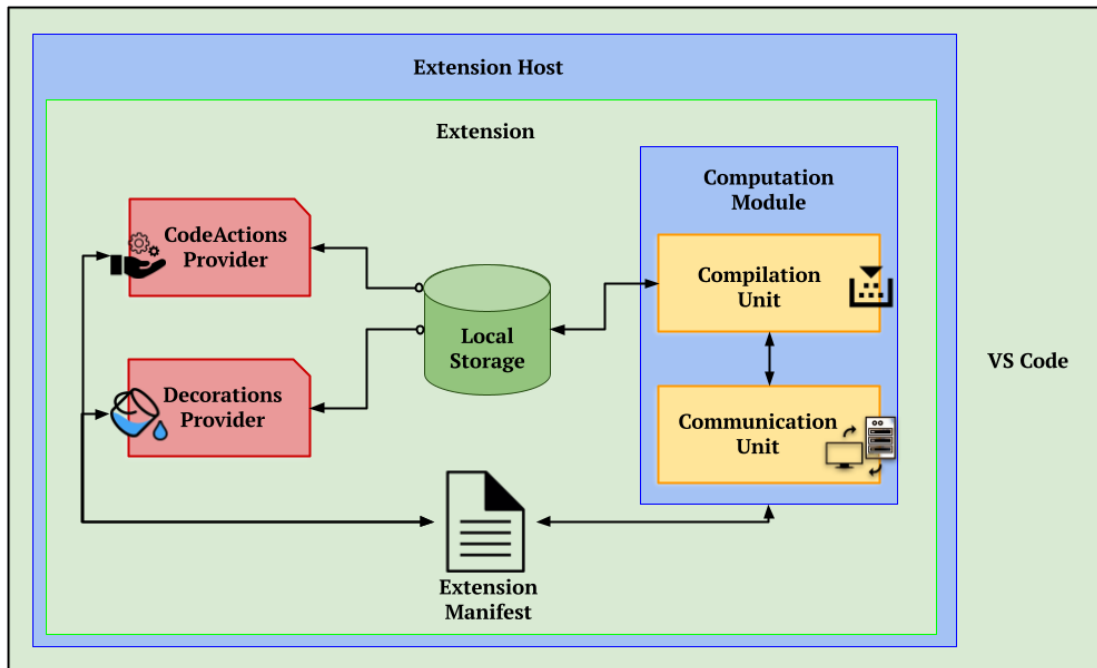


Figure 3.1: Architectural Overview of PYEDU frontend

As can be seen in Figure 3.1, the extension runs in the context of an extension host, a Node.js provided environment, which also exposes VS Code APIs for use in the extension. PYEDU frontend consists of four essential components to achieve its functionality viz., the Computation Module, the Code Action Provider, the Code Lens Provider and the Decorations Provider. The Computation Module is mainly used for program compilation and for communicating with the backend server. The Computation Module reads from and writes into the localStorage provided by VS Code. The other two components mainly read from the localStorage. All of the different modules/units are governed by the extension manifest. The purpose of these two components is to aid one of the two main functionalities of the tool i.e., to present the visual and textual feedback to the users, and also allowing provision for quick-fixes on the source code.

3.3 Extension Contributions

All of the various contributions made to VS Code by the extension were noted in brief, in subsection 3.2.2. This section provides details on the specific contributions made by the extension.

3.3.1 Commands

The extension contributes the following commands to VS Code:

1. **Toggle Code Actions:** This is the most important command among all. The extension registers this command for the purpose of displaying diagnostics on the Python program, that the user edits in the VS Code editor. This includes two steps.

³<https://code.visualstudio.com/api/references/extension-manifest>

⁴<https://VS Code-docs.readthedocs.io/en/stable/extensions/our-approach/>

- (a) Check if the program state is dirty i.e., if it has been modified since it was last seen. If not, simply fetch the diagnostics for the corresponding program from local storage and display in code actions. Otherwise compile the program to check for any syntax errors. If there are no errors, save time by doing nothing.
- (b) If there were compilation issues, send the source program. to a backend server that responds back with the diagnostic information. The returned information is then used to populate the code actions provider appropriately for displaying the diagnostic information.

Apart from providing the above functionality, each time the command is triggered, it actually toggles code actions provider, thereby turning it on or off. Whether to provide the functionality of code actions can also be controlled by a configuration property in the extension manifest.

2. **Toggle Decorations:** This is another command registered by the extension. The purpose of this command is to highlight sections of code in the Python program, that the user edits in the VS Code editor. The highlights are based on specific edits that need to be made to fix programming errors. The command first fetches the diagnostic information, similar to the above command, either from querying the backend server - if the program is in a dirty state - or directly from the local storage - if the program has not been modified since the last time it was seen. Upon fetching the diagnostics it populates the decorator class to highlight specific sections of the program.

Apart from providing the above functionality, each time the command is triggered, it actually toggles the decorations, thereby turning it on or off. Whether to provide the decorations feature can also be controlled by a configuration property in the extension manifest.

There are certain trigger events that govern when certain commands of the extension get executed. In our case, there are two ways in which the command execution is triggered

1. There are some icons that the extension contributes to the VS Code Menu, under the editor-title bar. Clicking on any of the icons triggers the respective command that it's linked to.
2. The extension also contributes some key-bindings. These define short-cuts, essentially key combinations, for triggering the execution of the commands that they are linked to.

3.3.2 Events

The extension also registers three workspace events that occur in VS Code. They are described below.

- **Document Save Event:** This event is triggered when a program/ document is saved. In handling this event, the extension communicates with the backend server through REST API calls by posting the source code of the program that is open in the active editor. The backend server responds to the client with all the diagnostic information that it can aggregate with the help of the augmented AI technique. It may be a case that the features to display this diagnostic information on the text editor may very well be toggled off, which means that

there may not be anything to actually display to the user. But still fetching the diagnostic information pre-emptively, through such events, will enable to quickly display hints and suggestions to the user in a responsive manner. In the process of handling this event, the extension also updates both, the decorations, as well as the code actions, by invoking the respective providers.

- **Text Document Change Event:** This event is triggered when a new/ different program/ document is opened in the current editor window. The extension updates the decorations in handling this event. It is to be noted that the code actions provider needn't be invoked for updating the code actions while handling this event, as it is maintained as a map of documents to corresponding diagnostics through the diagnosticCollection API that VS Code provides.
- **Text Editor Change Event:** This event is triggered when the editor context is changed i.e., the user moves out of the current active editor and switches to another editor window. Like in the above event, the extension updates decorations in handling this event.

3.3.3 Menu Icons

The extension contributes two icons to the VS Code Menu, under the editor-title bar. As can be seen in Figure 3.2, the monocular symbol and the highlighter symbol are the icons provided. Both of these are linked to commands that are registered under the extension. The monocular icon is linked to the command related to toggling the diagnostic, and other icon is related to toggling the highlight, both described in subsection 3.3.1. Clicking these icons trigger execution of the respective commands linked to the them.

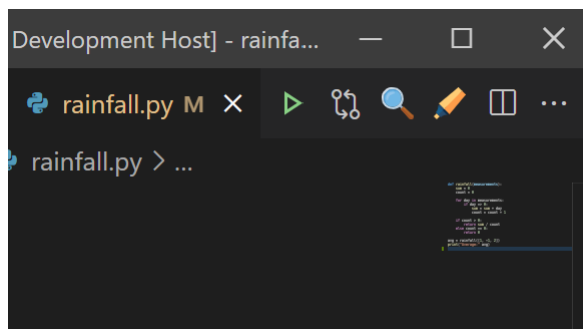


Figure 3.2: Menu Icons of the VS Code Extension

3.3.4 Key-bindings

The extension contributes key-bindings, essentially keyboard shortcuts for triggering the execution of some of the commands registered under the extension. Pressing the respective key combinations cause execution of the respective commands. These key-bindings are configured to be enabled only when the editor is in focus i.e., there is an active (text) editor in focus. What follows is a list of key-bindings that is defined by the extension.

Command	Windows Key	Mac Key
Toggle Code Actions	Ctrl + F1	Cmd + F1
Toggle Decorations	Ctrl + F2	Cmd + F2

Table 3.1: Extension Key Bindings

3.3.5 Configuration Properties

These are some of the fields/ properties that are defined in the extension manifest. The extension can programmatically read/ write the values of these configuration properties. Given below is a table of such properties.

Property	Type	Default Value
enableCodeLens	boolean	false
webServerPath	string	http://localhost:5000
diagnosticLevel	integer	1
activeHighlight	integer	0
enableDiagnostics	boolean	false

Table 3.2: Extension Configuration Properties

3.4 Feature Modules

In the following sections we give an overview of the individual entities depicted in Figure 3.1, highlighting the four main components that contribute to PYEDU frontend i.e., the VS Code extension.

3.4.1 Local Storage

The local storage, as the name suggests, provides the capability to store data that is accessible in the extension context. The VS Code API provides a Memento object for local storage. Memento is a storage utility that can be used to store and retrieve values and is exposed through the VS Code API. When an extension is activated, the extension context consists of two Memento objects, workspaceState and globalState, that are used to store state in the context of current workspace and independent of the current workspace respectively. We implement a Local Storage Service class that uses the workspaceState Memento object to store and retrieve data relevant to our application. This workspaceState is used to store a history of the latest diagnostic information obtained from the backend server, for every program that the tool has seen thus far. Whenever the tool processes a particular document (with the help of the extension), if there is no change detected in program/document state i.e., no modification was done since the last time the tool was run on the program, then the history of diagnostic information, corresponding to the particular program, is fetched from the workspaceState instead of invoking the backend server again requesting for fresh diagnostic information.

3.4.2 Computation Module

The computation module is logical division of the extension functionality and can itself be divided into two units viz., Compilation Unit and Communication Unit.

- **Compilation Unit:** The compilation unit is responsible for checking if a given source program compiles successfully or not. This is achieved with the help of `py_compile`, a module which is used to translate python source programs to bytecode. This is used prior to establishing any communication with the backend server.
- **Communication Unit:** The communication unit is responsible for all communication with backend server. It makes REST API calls by posting the user source program to the server and stores the returned responses in the local storage.

3.4.3 Code Actions Provider

Code actions are a language-specific editor feature provided by the VS Code Extension API. They are used to implement quick fixes and refactoring in VS Code. The diagnostics registered in the code actions are highlighted in the source code with squiggly under-lines. If a code action is available, it is announced by a bulb icon near the source code when the cursor is positioned on a squiggly line or a selected text region. The light bulb can then be clicked to reveal the Quick Fixes. If we click on the link to Quick Fix, it provides an option to automatically apply the suggested fix to the source code.

The extension registers a code actions provider. The code actions provider populates an array of code actions which correspond to the syntax error related diagnostics. These diagnostics are generated by PYEDU for any given source program that PYEDU has been invoked on. The code actions provider updates these diagnostics under the following two conditions.

1. When the command for toggling diagnostics is invoked, either through the click of the icon or through the keyboard shortcut, the code actions provider is invoked to update the corresponding document/ program's code actions.
2. When the document save event is triggered, then also the code actions provider is invoked to update the code actions corresponding to the document/ program associated with the event.

The diagnostic information received by the client in response to the query made to the backend server consists of an array of repairs. Each repair consists of the following fields:

- **Line number:** The line number in the source program which has found to be erroneous.
- **Repair Line:** This is the repaired version of the erroneous source line.
- **Repair Class:** Repair class indicates the type of repair needed (insert/ delete/ replace/ misc), along with the list of tokens to be inserted and deleted from the erroneous source lines to repair them.
- **Feedback Object:** A JSON object which incorporates the following items of feedback:

- **Full feedback:** A textual feedback message, containing all elements of the repair to be made.
 - **Partial feedback:** A textual feedback message, containing only some elements of the repair to be made.
 - **Repair Action:** Indicates the type of repair to be made i.e., insert, delete or replace.
 - **Tokens:** The tokens in the source line on which the repair action has to be taken.
 - **Token Description:** Textual representation of the tokens on which the repair action has to be taken.
- **Edit Diff Set:** An array of JSON objects indicating all the repair actions and the corresponding locations in the source line where they need to be applied.

Considering that the users of PYEDU, and the instructors who might use PYEDU in offering their programming course, might have different objectives (learning or otherwise), it is only imminent that we allow the extension to provide different capabilities. This is achieved with the help of a configuration properties that are set to indicate the level of feedback that is required to be presented. We currently identify two types of use cases, a novice programmer and an experienced programmer. A novice programmer could mean students or professionals who have just started their journey in learning to code. An experienced programmer could be an experienced developer or a student who has previous programming knowledge, either from the past or in the course of an on-going programming course.

The code diagnostics provided by each of the code actions contains the following key elements.

- **Diagnostic Message:** This is the message displayed to the user when they hover over the squiggly line indicated by the code actions.
- **Range:** This is the range of underline i.e., the range of text in the source text that is to be underlined to catch the attention of the user to that specific portion of the source line.
- **Severity:** The error indicated could be categorized into different types such as *warning* or *error*, based on the severity of the error. The default is set to *error*.
- **Source:** This is just a nice short text describing the origin of the code actions i.e., speaks about the source of the code actions provider (as there could be multiple code actions at the same location made available by different providers).

The code actions itself has a few options that need to be configured as well:

- **Code Action Type:** In our case it would simply be QuickFix kind.
- **Diagnostics:** A repeat of the diagnostic information from above.
- **Edit Operation:** The actual changes to be made to the source line for fixing it.

We highlight some of the design decisions regarding the visual feedback enabled by the code actions provider in the following paragraphs.

If the learning objective is very strict, so as to reveal very minimal details to the programmer, then the range of underline could either be nothing i.e., it is hidden or the entire source line that is erroneous. In general we would select the next alphanumeric word occurring after the position of edit described by the items of the Edit Diff Set.

The diagnostic message could be set according to the configured level of feedback. For instance, if it is for a novice programmer, we may actually show an abstract full text feedback from above, or a little less revealing feedback message based on the learning objectives set. An experienced programmer might just be interested in knowing the fix to be applied, so we could simply show a list of tokens expected or even the exact fix.

The code actions message might be a long text description prompting a novice programmer to think about the kind of fix that may be required to be made, without revealing the exact fix, or just a brief about the exact repair action to be performed in case of an experienced programmer.

The source line edit operations of the individual code actions edit might also be adjusted accordingly. For a novice programmer, there might, in essence, not be any automated repair made available, whereas an experienced programmer might simply want to get through with the syntax errors, typographical or otherwise, and focus on the more important aspects of semantics of the program, and the main task that the program must achieve.

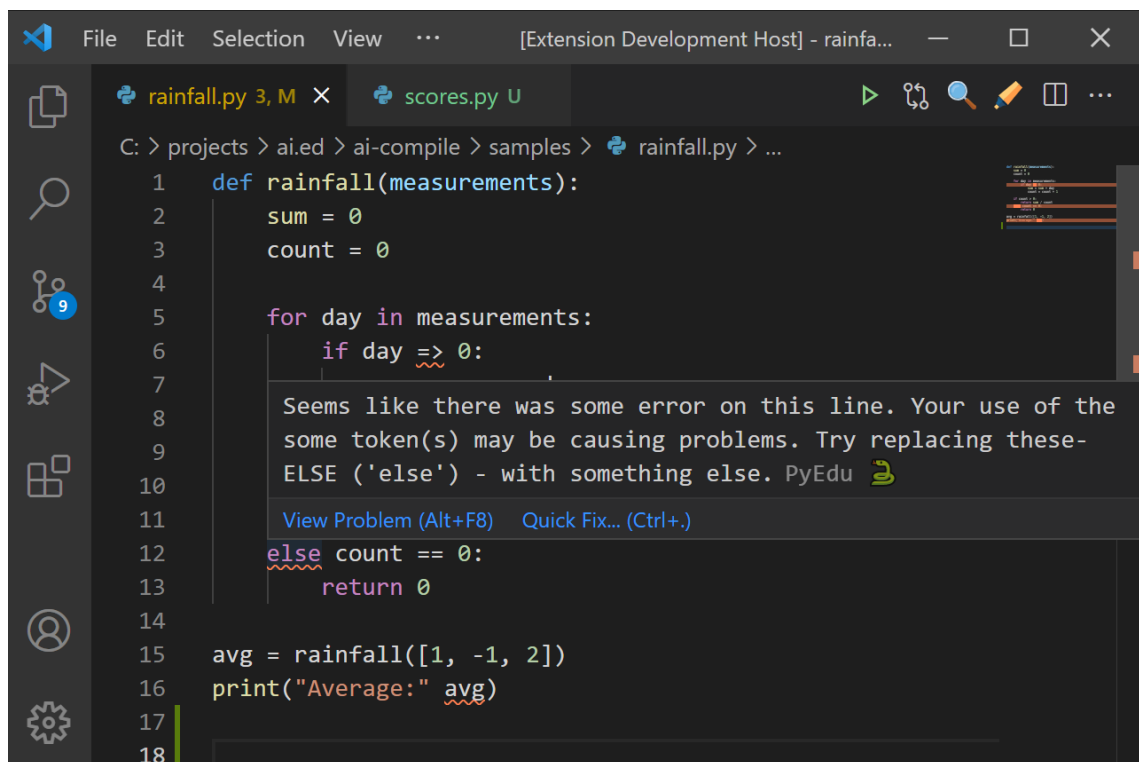


Figure 3.3: Code Actions Provider

3.4.4 Decorations Provider

A decoration is, as the name suggests, some style elements that can be applied to range of text in an editor. The Decorations API can be used to decorate pieces of code. It allows us to decorate ranges of text in an editor using a subset of CSS properties. Rather than modify the textual information,

decorations are simply a visual addition to viewing the code.

The extension registers some events that simply update decorations, when triggered. The decorations provider updates the active editor with decorations corresponding to the syntax error related diagnostics of the document/ program that is open within it. The decorations provider is implemented as a custom class, which mainly provides the functionality of updating the decorations of documents. The diagnostics are updated under the following four conditions.

1. The command for toggling diagnostics is invoked
2. A document is saved
3. The active document in the editor is changed i.e., a new/ different document is opened
4. The active editor is changed

Items 2, 3 and 4 above trigger the corresponding events where in the decorations update API is invoked to change the decorations appropriately, to correspond to the new/ changed document/ program.

We refer the readers to subsection 3.4.3 for a description on the detailed feedback received from the backend server.

Again, considering that the users of PYEDU, and the instructors who might use PYEDU in offering programming courses, might have different objectives (learning or otherwise), we should allow the extension behavior to change with the respective expected outcome. In such a scenario, a configuration property in the extension manifest can be set to indicate the level of support/ feedback required. As noted previously, we currently deal with two levels, one at the level of a novice programmer and other at the level of an experienced programmer.

The use of the decorations provided incorporate the following key elements:

- **Decoration Type:** Perhaps the most important from a look and feel perspective, these contribute to the aesthetics of the decoration by defining some styling elements through pre-set fields. We define more than one decoration type to encode decorations with different semantic meaning.
- **Range:** This is the range of decoration i.e., the range in the source text that is to be decorated to catch the attention of the user to those specific portions.
- **Hover Message:** This is a message that displayed to the user when they hover over the text in the range of decoration. We use this field to show the programmer relevant feedback corresponding to fixing the source line.

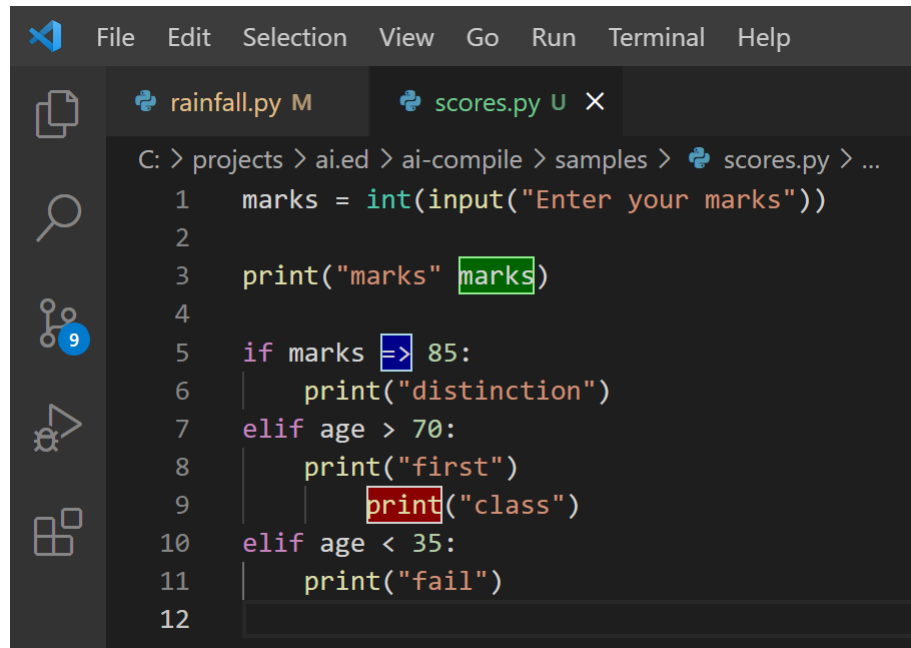
We highlight some of the design decisions regarding the visual feedback provided by the decorations in the following paragraphs.

In the case of a novice programmer, if the learning objectives are very strict, providing minimal hint/ help to the programmer, leaving much of the fix to be figured out by the user themselves, we simply highlight the entire source line that is erroneous. If it is required to give a little more information to the programmer, we highlight only sections of the source line that are of special

interest, those corresponding to the actual locations in the source line where repair action has to be applied.

In the case of an experienced programmer, we may provide more informative feedback to the user by not only decorating the relevant sections of the source line, but also color-coding them by decorating different kinds of repair actions (insert/ delete/ replace) with different decoration types. The hover message, that which contains the feedback shown to the users, is set according to the level of feedback configured. With the help of the responses received from the server, as described in subsection 3.4.3, there are different kinds of messages that could be displayed to the users:

- **Full feedback:** We could show the programmers a textual description of the complete feedback returned from the server, that includes the tokens that need to be inserted/ deleted.
- **Repair Classes:** We could only show a list of tokens that need to be inserted/ deleted.
- **Repair Line:** We could simply indicate the exact fix that needs to be made by displaying the fixed source line.

A screenshot of the Visual Studio Code editor interface. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, and Help. Below the menu, there are two tabs: 'rainfall.py M' and 'scores.py U X'. The main editor area shows a Python file named 'scores.py' with the following code:

```
C: > projects > ai.ed > ai-compile > samples > scores.py > ...
1  marks = int(input("Enter your marks"))
2
3  print("marks" marks)
4
5  if marks => 85:
6      print("distinction")
7  elif age > 70:
8      print("first")
9      print("class")
10 elif age < 35:
11     print("fail")
12
```

The code is decorated with various elements: a green box around 'marks' in line 3, a blue box around '85' in line 5, and a red box around 'print' in line 9. The left sidebar shows icons for Explorer, Search, Source Control, Run and Debug, and Extensions.

Figure 3.4: Decorations Provider

3.5 Application Flow

In this section we describe a typical way in which any user would use PYEDU. Figure 3.5 summarises the workflow of PYEDU. Throughout this chapter we have mainly highlighted the client side application of PYEDU in the form of the VS Code extension. Figure 3.5 shows the flow of the application, including the aspects of the backend server involved.

1. In order to make use of the various functionalities of the extension, it is first of all required to activate the extension. In order to activate the extension, the user would need to open a

Python program in the VS Code editor. After the extension is activated and loaded, the user will be able to see two icons appear in the editor-title bar, one for displaying diagnostics, and another for displaying decorations.

2. Every time the user saves a Python program, the extension goes through a compile-fetch-update cycle. In the compile-fetch-update cycle, the source program is firstly compiled to check for any syntactic errors. If there are any, then the source program text is sent to the backend server, to fetch diagnostic information on fixing these errors. After receiving a response from the server, the extension updates the display elements, including the code actions and the decorations.
3. The same compile-fetch-update cycle is also triggered when the user clicks on either of the extension icons. The icons serve as a means to toggle the respective diagnostic display features on or off. The user can also trigger the compile-fetch-update cycle by invoking the respective commands via the keyboard shortcuts that are defined in the extension manifest.
4. While the above defined events and commands update the code actions diagnostics, the code decorations is also updated with a couple of other events. Whenever the user opens a new document or when they navigates to a different editor, the user defined event handlers for the same update the decorations. This is required due to the way the decorations API is implemented. Whereas, in case of code actions, the provider maintains a document/program wise record of the diagnostics, the same is not possible for code decorations.
5. When the code actions diagnostics is enabled, the user can see the corresponding diagnostic information on the buggy source lines that PYEDU was able to identify. When the user positions the cursor on the squiggly lines, the user will see the linked diagnostic information and also a link to QuickFix defined for it. The user will also see a light bulb pop up, clicking which takes them to the available QuickFix. The user can then click on the displayed QuickFix to automatically apply the fix to the respective source lines.
6. When code decorations is enabled, the user can see the corresponding visual highlight (based on the decoration type) and diagnostic information on the buggy source lines that PYEDU was able to identify. When the user hovers over the text regions that are decorated, he will see the linked diagnostic information.

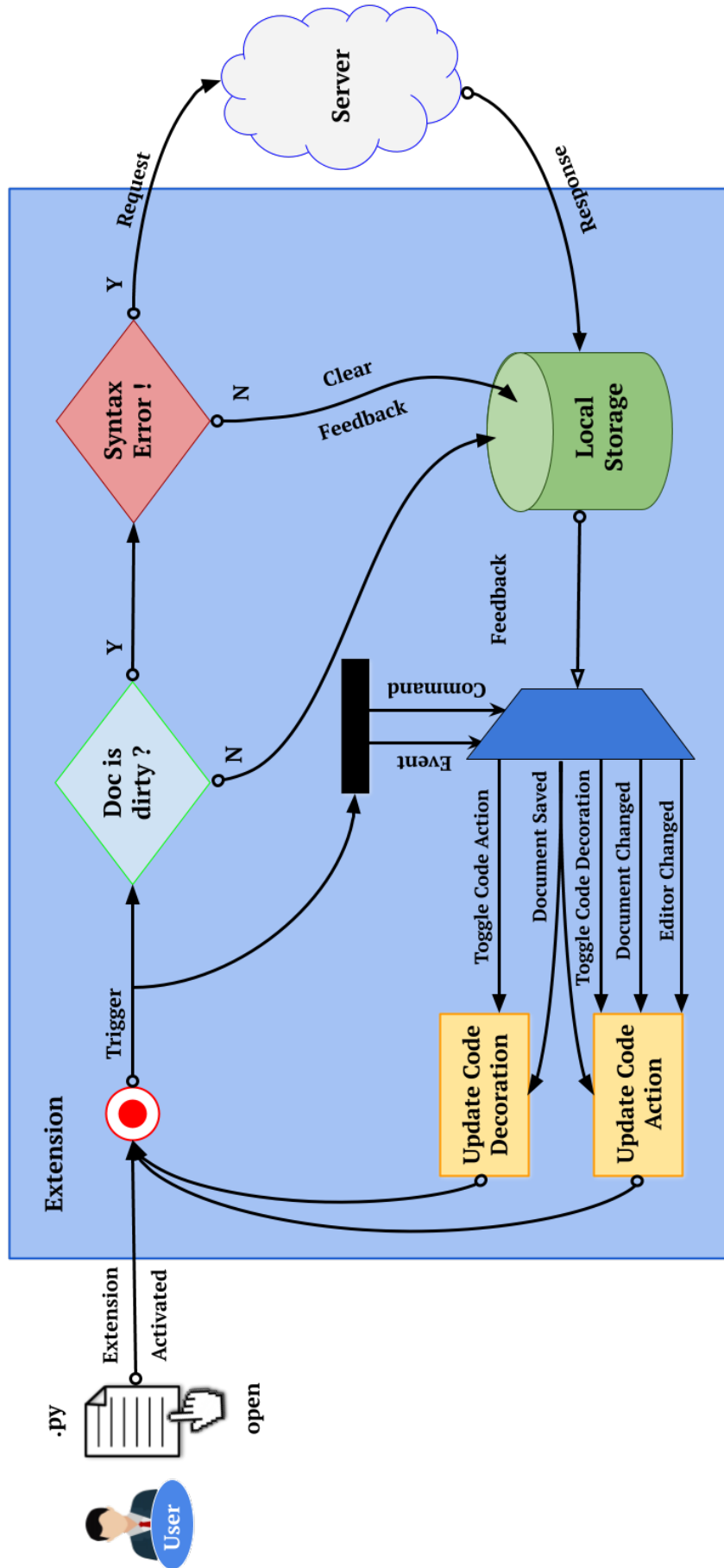


Figure 3.5: Flow of PYEDU Application

Back-End Server

Contents

4.1 Overview	35
4.2 API Description	36
4.2.1 Response from the API	37
4.3 Illustration	38
4.4 Unit Tests	38

4.1 Overview

In this chapter we describe the design and development of the back-end server for PYEDU. The server end comprises of the communication server that handles all requests from the client and the AI-engine that sits as the intelligence for generating responses relevant to the client request. The communication server queries the AI-engine and forwards the response received to the client side.

As described in chapter 2, we use PyMACER, a light-weight modularized machine learning technique that predicts fixes for syntax errors in computer programs, for the AI engine that augments the backend. Before we delve into the design and implementation aspects, let us take a look at why a technique like MACER would be most suited as the AI-system to augment the backend. Before presenting a highlight of points to follow, we refer the reader to chapter 2, which describes MACER and PyMACER in detail. The following points highlight some of the utilities of a such a technique in helping returning useful feedback hints to the client.

- It is observed that the repair line might actually differ from the compiler reported line number, even in case of interpreted languages like Python. For instance a missing parenthesis on one line might only be caught as a syntax error reported on subsequent lines. MACER follows the TRACER strategy for picking candidate error lines (lines adjacent to the compiler reported error line) helps catch more errors than would otherwise have been possible.
- The repair class itself provides a wealth of information. It indicates, in abstract form, what tokens need to be inserted and what tokens need to be deleted, in order to fix the compilation error. This could be one way of prompting hint to the student, without actually revealing the exact fix. A concretized version of the tokens could also be prompted as necessary.
- The repair profile that MACER predicts is also very helpful in visually highlighting the location within the line where the fixes need to be made. This could be used in conjunction with some other details as mentioned previously or even independently.

- The exact fix recommended by MACER could itself be a last resort help that could be made available to the students, under specific circumstances.

As discussed earlier the main task of predicting fixes for the erroneous lines is done by the machine learning technique described in chapter 2 on PyMACER. Hence, the goal of the back-end server is simply to provide an API endpoint which would in turn invoke the machine learning model, and return the output of the machine learning model to the caller. We created a REST API using the Flask¹ package.

The following section describes the API in detail. We also elaborate on the format of the request and the responses exchanged between the client and the server. To clarify things further, we explain the process with the help of an example illustration. We finally speak about an important aspect of testing the API and the unit testing that was done to ensure the reliability of the developed API.

4.2 API Description

Backend API: Get Program Fixes	
URL	/getfixes
HTTP Method	POST
Action	The API takes a single parameter source , the source code text of the user program. The server invokes PyMACER and passes on the received source code. PyMACER generates the correct program and returns the relevant information back to the server. This is then encoded as a JSON object and returned to the caller.
Return Data	A JSON object consisting of an array of repairs, one for each erroneous line. Each repair has the lineNo, repaired line, a list of operations needed to repair the line, and a feedback object.

The REST API is as described in the above table. As mentioned, the API returns a variety of information on top of the fixed erroneous lines. This information can very well help the client side application better describe the error to a student and also progressively reveal hints in order to maximize the students' learning and meet the learning objectives. We further elaborate on the various aspects of the information returned by the API in the following section.

¹<https://flask.palletsprojects.com/en/2.0.x/>

4.2.1 Response from the API

The response from the API is a JSON object having the following key-value pairs:

- **repairs:** A top-level key consisting of an array of values corresponding to the fixes/repairs for each erroneous line.

Repair for each line has the following information.

- **lineNo:** A number indicating which line the repair corresponds to.
- **repairLine:** The fixed line which is free from errors. Displaying the repairLine directly to a student may not be pedagogically instructive. But in case if the user is a seasoned programmer this kind of feedback can increase their productivity.
- **repairClasses:** A list of repair classes applied to this line in order to repair it.
- **feedback:** A JSON object having details which can be used to provide additional feedback to the user.
- **editDiffs:** A list of JSON objects indicating where edits (insert/ delete/ replace operations) are required to transform the erroneous line into the correct one. This can again help in highlighting relevant parts of the erroneous line to pin-point exactly where something went wrong.

The API also returns a feedback object which contains additional information that can be utilized by the front-end to provide more informative messages to the user. The feedback object contains the following pieces of information:

- **Fulltext:** A textual feedback message that can be shown to the user. This feedback is automatically generated as part of the PyMACER pipeline based on the repair classes predicted. This message can be broken down into various sub-parts.
- **msg1:** A generic message to be shown to the user, indicating that there are some errors in the given line.
- **msg2:** A more specific error message hinting at what kind of edits are required to fix the given line.
- **actionMsg:** A message pinpointing exactly what operation (insert/ delete/ replace) has to be performed to fix the given line.
- **tokens:** The tokens in the erroneous line on which the action has to be performed, for instance (or).
- **tokensText:** Textual description of the tokens on which the action has to be performed, for instance CLOSE_PAREN or OPEN_PAREN.

4.3 Illustration

```
1 name = int(input("Enter your name"))
2 age = int(input("Enter your age"))
3 print("Hello", name)
4 if age >= 18:
5     print("Can vote")
6 else age < 18:
7     print("Can't vote")
```

In this section we take an example source program, shown above, and describe the response generated by the REST API. Please note that there is an error in the above Python program on line 6, where the user has used *else* in place of *elif*.

When we submit this program to the REST API, it returns a JSON object as shown in the following page. As we see the back-end correctly identifies that there is some problem at line number 6 and returns the corrected line. The repair class indicates that we need to delete an 'ELSE' keyword and replace it with an 'ELIF' keyword which is indeed the required correction. Moreover, the feedback returned by the back-end focuses not only on what went wrong but also on what might be needed to correct the program. This type of feedback can be valuable to a novice programmer. The *editDiffs* indicates the column numbers where the edit needs to be made which can be used by the fronted to better pin-point where the student made an error.

4.4 Unit Tests

For any software, it is important to write unit tests since they ensure that the code meets quality standards and helps catch bugs and issues at an early stage in development. Since the back-end has only a single API we wrote 2 unit tests for testing how the API behaves in two scenarios - when the request is valid; when the request is invalid. It is to be noted that, for this API, an invalid request means that the source code doesn't contain any errors. This also includes the case when the source code is empty. In case of the correct request we check that the response contains the top level JSON object "repairs" and has all the sub-keys described in the earlier sections. For the invalid request case, the response should still contain the top level object but the value should be an empty list. We use `pytest`² to make sure that the code passes all the unit tests before it is deployed.

²<https://docs.pytest.org/en/6.2.x/>

```
{
  "repairs": [{
    "lineNo": 6,
    "repairLine": "elif age < 18 :",
    "repairClasses": [
      "- ELSE\n+ ELIF"
    ],
    "feedback": [{
      "fullText": "Seems like there was some error on this line.
        Your use of the some token(s) may be causing problems.
        Try replacing these - ELSE ('else') - with something different.",
      "msg1": "Seems like there was some error on this line. ",
      "msg2": "Your use of the some token(s) may be causing problems.",
      "actionMsg": "Try replacing these",
      "action": "Replace",
      "tokens": [
        "elif",
        "else"
      ],
      "tokensText": [
        "ELIF",
        "ELSE"
      ],
      "misc": ""
    }],
    "editDiffs": [{
      "type": "replace",
      "start": 0,
      "end": 3,
      "insertString": "elif"
    }
  ]
}]
}
```


Conclusion and Future Work

In this part of the thesis we looked at the design and development of PYEDU, a tool which interfaces with students as a VS Code extension for helping them fix programming errors in the Python programming language. This is made possible by presenting helpful visual feedback on syntax errors.

The PYEDU tool, although very much usable, is still in its nascent stage. While there could be some changes incorporated in the front-end as well, much of the improvements to the system could be thought of at the back-end side of things.

On the extension front, the feedbacks displayed right now are rather generic. To make it more beneficial for pedagogical application, the fixes to be made could be categorized into different groups, each mapping to learning of a particular programming concept, which could then help instructors/ developers design better feedback prompts to the students. This would require a good amount of support from the back-end, for it to return additional information, in terms of the context of the repair being made. We could also enable the extension to collect additional information from its usage for instance by logging snapshots of students' programs as they learn to code.

On the back-end side, the AI-system that generates fixes for python syntax programming errors is what could be enhanced mainly. While there is always a possibility of replacing the AI-system with another, the options are rather bleak at the moment. Instead the existing technique itself could very well be improved and is actually work in progress.

Currently PYEDU works in a client-server context. In a future iteration it may be beneficial to handle the entire processing of the system in the extension context itself, this would obviate the need for a central powerful server, especially in the context of a large student base that is trying to access the system simultaneously.

Part II

PRIORITY

Introduction

Contents

6.1 Background	46
6.2 Our Contributions	46

Use of technology in aiding education isn't just recent but has seen itself being increasingly used over the past few years. This is especially true of courses offered by premier educational institutions. There are three main stake holders to any such course viz., the instructor(s), the tutor(s) and the student(s) who are enrolled into the course. In many such courses, the tutors have a vital role to play, alongside the instructors.

For our particular use-case we consider ESC101, an introductory programming course offered to freshman students at Indian Institute of Technology Kanpur. One of the responsibilities of the tutors of this course, involves preparing programming assignments or problems for the students. Every offering of ESC101 requires that the tutors set questions for labs and exams afresh, starting from scratch. This is a laborious process and often just re-inventing the wheel, as the syllabus of the course seldom changes and evolves only very gradually. Yet there has been absolutely no use of such historical problems data and the tutors have to start creating problems afresh in every iteration of the course. This leaves behind a tremendous scope for making use of such historical data, which serves to assist the tutors of the course. What we are essentially trying to achieve is to build a system of program retrieval for enabling useful access to such a large corpus of data, through some sort of labels or tags affixed to each of those programs. The following are some of the motivations behind building such a system:

1. There have not been any efforts in this direction in the past, in the field of program retrieval (based on labels). This provides scope to demonstrate usefulness of such a system.
2. There is a large, unused, corpus of problems data available from the past offerings of ESC101, which is only a dump of data right now and not been put to use at all. Building a problem search portal facilitates its usage for making the tutors' work easier.
3. Building such a portal would also facilitate students themselves to use the system for practice sake.
4. It also provides for further research in this field, on measuring the efficacy of using such aids in assisting conduction of such courses etc.

The scope of this part of the thesis is to develop a first generation web application portal for program retrieval, by enabling access to such a corpus of data in the back-end. The web application must be easy to access, user-friendly and provide an intuitive interface. It must also be developed in such a way that it can easily accommodate new features atop the existing ones.

6.1 Background

In this section, we familiarize the readers to the ESC101 course and the PRUTOR platform in a little elaboration, as the development of the entire web application centres around them.

ESC101 ESC101 is an introductory programming course offered by the Department of Computer Science and Engineering, at Indian Institute of Technology Kanpur, for freshman undergrad students. This course is intended to familiarize students with the basics of computer programming. There are weekly lab assignments and practice problems made available to the students for practice. Interim tests and exams are also conducted as part of this course. The students are evaluated on their performance on each of these. The tutors of the course are responsible for creating programming questions, be it for weekly programming assignments, for regular practice or as questions for tests/ exams. This course has been offered a number of times in the past and the domain and scope of problem creation remains the same. The repository of questions are available as a dump of Prutor data [6], the platform which is used to offer this course.

PRUTOR Prutor, a tutoring system, is a web based IDE and the platform used for conducting introductory programming courses at Indian Institute of Technology Kanpur. Prutor is a cloud-based web application that provides instant and useful feedback to students while solving programming problems. Prutor stores, at regular intervals, the snapshots of students' attempts to solving programming problems. These intermediate versions of the student programs provide the instructors (and data analysts) a view of the students' approach to solving programming problems. Prutor also facilitates tutors of the ESC101 course, to submit programming assignments (from labs and exams), created for the students of the course. It also enables collection of all such problems created by the tutors and stores the data dump in the back-end.

6.2 Our Contributions

In this part of the thesis, we design and build a simple, intuitive, user-friendly and efficient system for program retrieval. In specific, the following points highlight the major contributions made by the thesis.

Front-end Design and Development The user interface is developed by keeping the non-functional and functional requirements of the users in mind. With students and faculty members as the end users, the goal is to keep it as simple and lightweight as possible. Aside this, the front-end was also designed so as to facilitate a handy collection of usage statistics (feedback) of the portal. The front-end was built using the Angular framework. Some of the major functional requirements include:

- Provide a login based authentication system
- Provide login/ logout functionality
- Display relevant results based on a user query

- Register page navigation events and user click events
- Register user submitted Feedback etc.

Some of the major non-functional requirements include:

- Provide a lightweight/ responsive user interface
- The user interface must be easy-to-use by the students
- Make the web app available all the time, and for all its users
- Make incorporating additional features atop the existing ones easy to do etc.

Back-end Design and Development In addition to the major front-end component, we also design and develop a Django back-end with MySQL running as the database server. It details on the various configurations to enable a mildly secure and an easy to maintain web application. We also report some of the optimizations that needed to be made in order to procure the most relevant set of results for the display component (front-end). The set of APIs that were designed and implemented is elaborated in section section 8.3.

Web Application Deployment The thesis also highlights some of the design choices made with respect to the actual deployment of the web application portal. In a nutshell, the front-end and back-end have been hosted as two separate docker containers on a cloud compute server¹ instance at Department of Computer Science and Engineering, Indian Institute of Technology Kanpur.

The developed web application portal is named PRIORITY after PProblem IndicatioR Reposi-TorY. The following chapters explain the previously mentioned contributions to the PRIORITY system in detail. Chapter 7 presents the front-end architecture of PRIORITY and elaborates on the key components of the user interface. Chapter 8 presents the back-end architecture of PRIORITY and elaborates on the key components for data storage and retrieval. The deployment of the web application portal is described in chapter 9. It also documents results from a user-group study, who used the portal in a live environment, in an offering of the ESC101 course to freshman undergrad students at the Indian Institute of Technology Kanpur, for the even semester of 2020-21.

¹<http://pir.cse.iitk.ac.in/>

Web-Application Portal

Contents

7.1	Goals of PRIORITY portal	50
7.2	Architectural Overview	51
7.2.1	Angular Application	51
7.2.2	PRIORITY Frontend	53
7.3	Feature Modules	53
7.3.1	Conceptual Units	53
7.3.1.1	Local storage	53
7.3.1.2	URL router	54
7.3.2	Authentication Module	54
7.3.2.1	Login Service	55
7.3.2.2	Login Component	55
7.3.3	Display Module	56
7.3.3.1	Problem Service	56
7.3.3.2	Auxiliary Service	56
7.3.3.3	Problem Search Component	56
7.3.3.4	Problem Display Component	58
7.3.4	Feedback Module	58
7.3.4.1	Feedback Service	60
7.3.4.2	Rating Component	61
7.3.4.3	Label Selection Component	62
7.4	Application Flow	63

Abstract *In this chapter we take a look at the front-end architecture of PRIORITY, a web application portal for helping students and admins of a programming course. The chapter begins with going over the key goals and objectives of building a web interface for PRIORITY. This is followed by an overview of the main components of the Angular framework. We then briefly explain the overall architecture of the system. Thereafter we take a look at the conceptual units that form PRIORITY, followed by elaboration on the individual modules, components and services of the system, and their interaction with each other. The chapter concludes by showcasing a flow of the application, a typical usage scenario keeping the end-users in mind.*

7.1 Goals of PRIORITY portal

PRIORITY is a web application portal developed for assisting tutors and other administrators of the ESC101 course. The portal has three main components to it viz., the front-end web interface, the back-end database and the AI-subsystem that powers it. The portal puts into use a large corpus of unused problems data, available as a Prutor dump, from the previous offerings of the ESC101 course. It does so by exposing an indexed database of the problems data via a web interface. The problems are indexed based on labels, which correspond to the programming concept(s) that is(are) involved in solving a particular problem, the level of expertise required to solve it and the difficulty level of the problem.

PRIORITY essentially helps the stake-holders have a question bank that is search-able. We could think of PRIORITY as a Google for ESC101, with a role to do just what search engines do i.e., provide users the most relevant, the most appropriate responses to their queries. In this respect the goals of this system is also in line with the latter, except it is not meant for commercial applications (such as advertising etc.).

The target audience for this portal and the way such a portal would benefit them is mentioned below:

- **Tutors:** Every offering of ESC101, the tutors are responsible for creating programming questions for the students of the course for solving in weekly assignments as well as in the exams. The portal is meant as a tool to help the tutors get fresh ideas on creating new questions and not spend too much time on building them from scratch.
- **Students:** As an extension, the portal could also serve as a tool for students of ESC101 themselves, for practicing questions, especially close to the their tests and exams. Making available, a search-able database of past questions helps serves this purpose very well.
- **Administrators:** Course administrators will also be able to monitor the usage of the portal, from tutors and students alike. The portal would help collect and visualize usage statistics including things like - what categories of labels are the students (and tutors) searching more often; when is the usage of the portal high; the number of users that access the portal for any given duration etc. Each of these stats can reveal umpteen number of useful details.

Keeping the target audience and the above objectives in mind, we went about building a first generation web interface, with scalability, simplicity and user friendliness as the key goals. The front-end is developed using the Angular Framework. The web interface of PRIORITY serves two main purposes:

1. The interface provides a way for users to easily query for programs based on a set of labels/categories. It displays a list of suggestions (clickable problem titles) fetched from the back-end, based on the user query. It also enables displaying individual problems in an elegant manner.
2. The portal also incorporates a feedback data collection system which procures both implicit and explicit feedback from the users of the portal. This information is extremely valuable and helps improve the AI-system that powers the backend.

Apart from this, the portal also incorporates a login/ logout system for the users, thereby only authenticated users can access its capabilities. We also refer the reader to chapter 6 which gives some more background to PRIORITY.

7.2 Architectural Overview

7.2.1 Angular Application

Angular framework is a preferred choice for developing feature-rich web applications. Angular apps are built with the Model-View-Controller architecture. The following are some of the biggest advantages that we found with using Angular:

- **Thorough documentation:** The extensive, up-to-date documentation of Angular and a wide community support makes it a very good choice as a front-end development framework. A robust development framework, Angular provides functionality out of the box and makes it very useful for creating simple applications.
- **Built with TypeScript:** TypeScript, being a strongly typed language, helps developers write clean and understandable code. It facilitates early discovery of errors and makes it easy to fix them too. TypeScript provides a lot of additional features on top of JavaScript.
- **Two-way data binding:** It enables two-way data binding, in that any change in Model data is reflected in the View too. This way there is no need to write additional code to ensure such synchronization.
- **Angular CLI:** This is a great tool that can be used to generate common blocks of code from the command line. This helps keep the code consistent and seamless in terms of updates as well.
- **Angular Material:** Angular Material provides a good collection of built-in UI components and modules which are ready to use. It offers a wide variety of UI components, such as navigation patterns, form controls, buttons etc., all based on Google's Material Design guidelines.

All the above advantages and more made it an easy choice for using Angular to build the web application portal front-end. Building Angular applications involves creating HTML templates possibly with some Angular mark-ups, building component classes that manage these HTML templates, incorporating application logic in services and encapsulating various components and services in what are referred to as modules. The application is launched by bootstrapping the root module. What follows is a brief note on the architecture of an Angular Application.

We give a briefly note on some of the key components of the Angular Architecture depicted in Figure 7.1 below:

- **Modules:** Modules describe the organisation of the different parts of an application. Angular apps are modular. Every Angular app has atleast a root module, typically named AppModule, that is bootstrapped to launch the application. Other modules, if they exist are called feature modules. Generally a small application just has the root module.

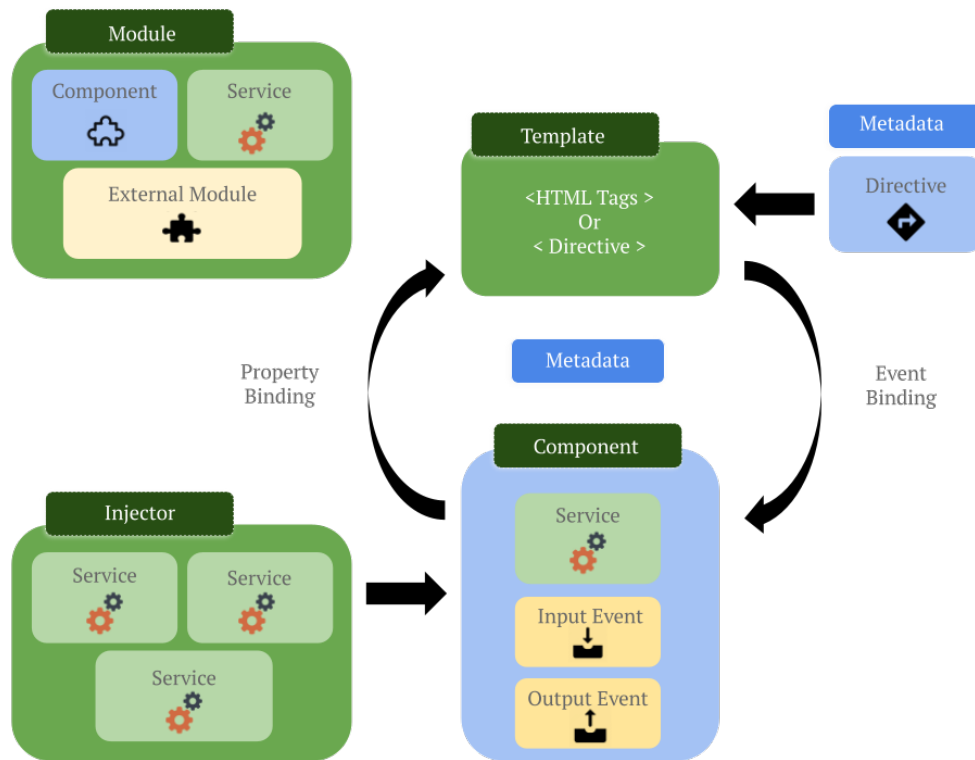


Figure 7.1: Angular Application Architecture

- **Components:** Components are the basic building blocks of the user interface for an Angular app. An Angular app contains a tree of Angular components. Every component is associated with an HTML template and contain a TypeScript class that defines its behavior.
- **Templates:** The component's view is defined by an HTML template. Templates tell Angular how to render the component.
- **Metadata:** These are special decorators used in a class that tell Angular about processing a class.
- **Services:** These are singleton objects instantiated only once during the entire lifetime of an Angular app. Services are mainly used to organize and share business logic and data with different components of an Angular app.
- **Dependency Injection:** Dependency Injection is a way to provide components with all services that they need. While creating a component, Angular asks injector, which maintains a container of service instances that were created earlier, for all the services that the component requires. If a service instance isn't found, it is created and added to the container. This way all required services are resolved and returned.

For more elaborate details on the Architecture of an Angular application we refer the reader to the official documentation¹.

¹<https://v2.angular.io/docs/ts/latest/guide/architecture.html>

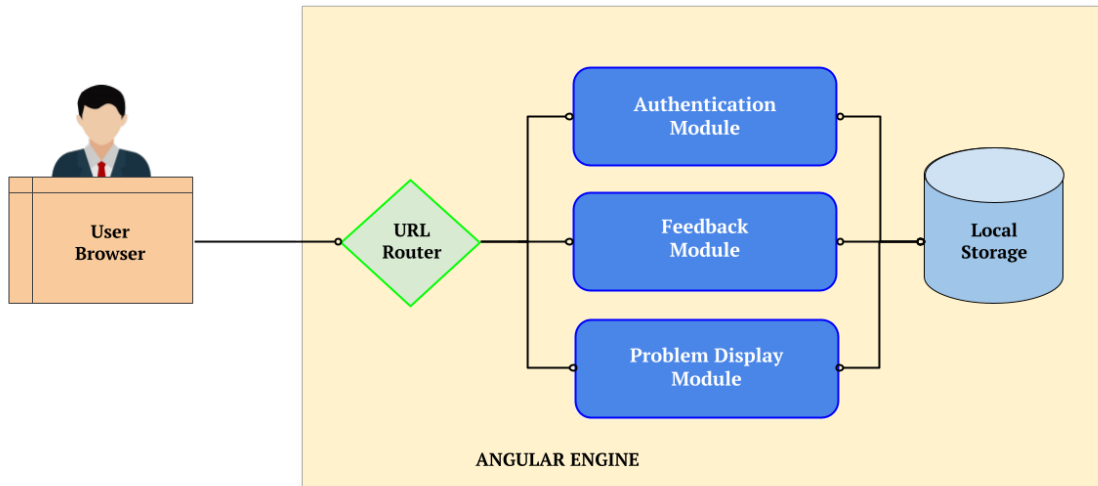


Figure 7.2: Architectural Overview of PRIORITY Frontend

7.2.2 PRIORITY Frontend

The PRIORITY front-end system is divided into three main feature modules viz., Authentication Module, Feedback Module and Problem Display Module. Each of these modules provide functionalities specific to a particular feature of the system. A modular segregation of various components and services ensures scalability and makes the system more maintainable.

The user tries to access the portal² on a web browser. The user request has to pass the URL router's validation and upon successful authentication, is appropriately directed the specific module of interest. The particular module then loads up the relevant application component to display the corresponding view. The different modules implement application logic specific to user requests. The modules further communicate with the back-end server through REST APIs calls and fetch data necessary to serve their specific functionality. The localStorage functionality provided in the browser context is used to save user data.

7.3 Feature Modules

In the following sections we explain in detail the individual entities depicted in Figure 7.2, highlighting the various feature modules that contribute to the PRIORITY frontend.

7.3.1 Conceptual Units

7.3.1.1 Local storage

The web browser provides localStorage as a way to store client-side data. It allows saving key-value pairs in the context of a web browser. Session management is handled with the help of localStorage. By having a client-side system for maintaining sessions, there is no server overhead. Client-side session management keeps the system response very fast and scalable, thereby improving user experience.

²<http://pir.cse.iitk.ac.in>

7.3.1.2 URL router

The URL router is responsible for mapping URL-like paths to views. It redirects the users appropriate views accordingly. A user validation system provides the route guard for controlling user access to various parts of the system. Any unauthorized/ unauthenticated user must not be able to access the various views of the system. Access to different modules, other than the login view, would require the user to be authenticated, if not the user is displayed the login view. If the user is already logged-in (is part of a valid session), then they would be automatically redirected to default search view of the portal, which displays a set of labels for the user to select while making a query to the system.

The users of the system are currently limited to one role, mainly the tutors. For every route request that a tutor makes, the session is validated with the help of localStorage. If the tutor is authenticated against the stored session information. As localStorage has no expiration date for the information that it stores, we implement a layer of logic that effectively deletes the stored information at regular intervals. This is specific to the particular tutor, and deletes stored information upon reaching a preset expiration time limit (when the tutor tries to access the system).

7.3.2 Authentication Module

PRIORITY is a web application meant for academic use and is accessible by limited audience, specifically some stakeholders of the ESC101 course. In order to prevent anyone from accessing the PRIORITY portal services, we have a user authentication in place, that is managed with some user session validation at the frontend in conjunction with the user authentication with the backend.

The authentication feature module is responsible for authenticating the users and managing sessions. The set of components, models and services it encapsulates provide functionalities for user login/ logout of the system in conjunction with user session management. This module is also responsible for preventing unauthentic users from accessing the system.

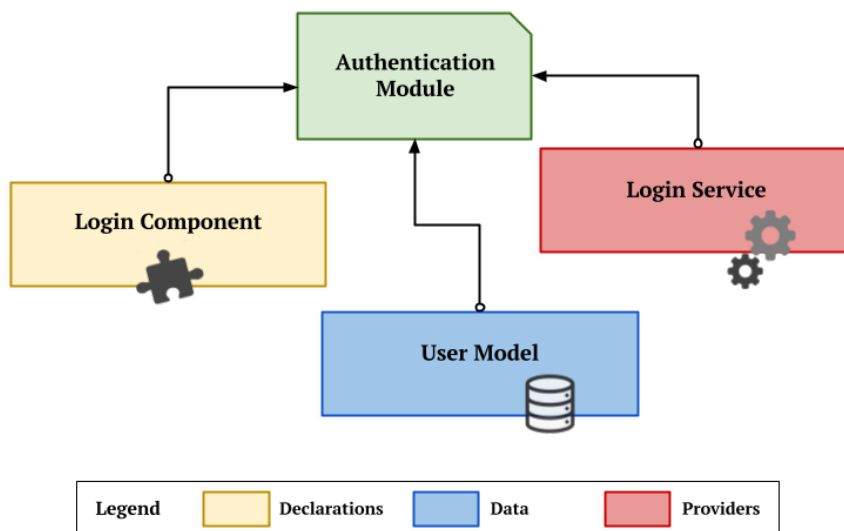


Figure 7.3: Authentication Module

7.3.2.1 Login Service

This service facilitates the functionality of user authentication and user session management. The service is provided at the root application injector. It provides the necessary method APIs for interacting with the backend APIs for performing user authentication. It uses the localStorage capability of the browser for the purpose of session management, as explained in subsection 7.3.1.2.

7.3.2.2 Login Component

This is the main component in this module and corresponds to the login form view rendered by the application. We make use of some of the available Angular material components for building this component. It contains various fields & methods that provide application logic for user login to the system and provides all of the necessary method APIs. The login view comprises of a simple login box and fields for entering the username and password as shown in Figure 7.5. Clicking on the sign in button logs the user in to the portal upon successful authentication. Pressing the sign up button just prompts a dialog box displaying the admin e-mail on which the user could mail, requesting for an account to access the portal. This is because, currently, user account creation isn't automated and happens after manual verification only.

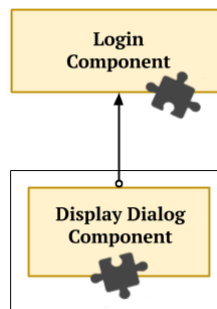


Figure 7.4: Login Component

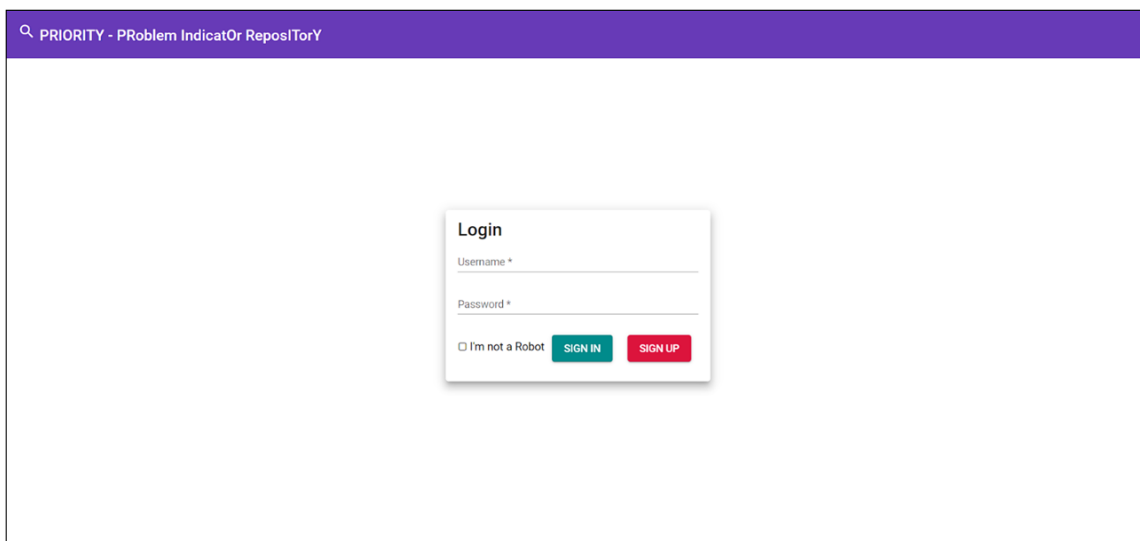


Figure 7.5: Login Component View

7.3.3 Display Module

The problem display module is the most important module of PRIORITY frontend and is central to serving its core purpose. This feature module provides the main functionality of displaying search results as well as displaying individual problem description, along with the associated adjunct capabilities. The module encapsulates two key components viz., the problem search component and the problem display component.

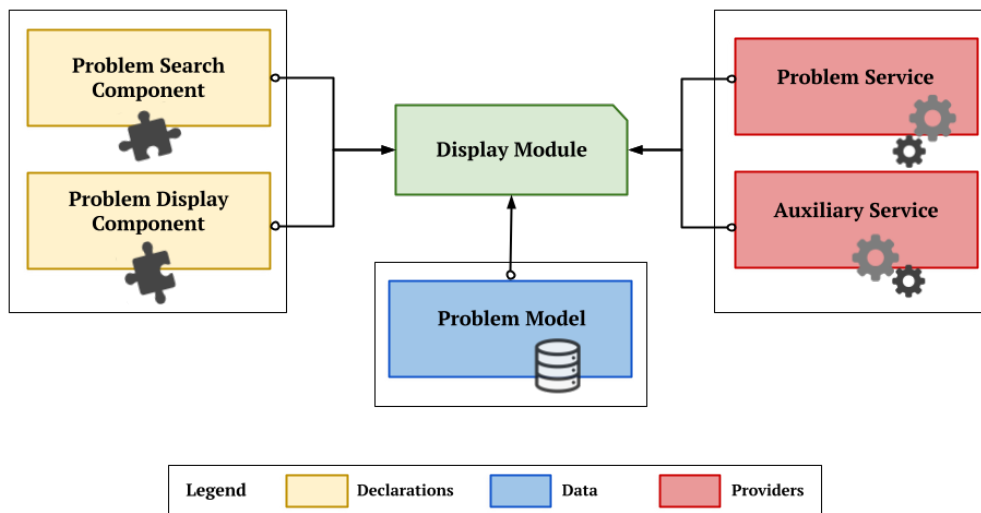


Figure 7.6: Display Module

7.3.3.1 Problem Service

The Problem service is provided at the root injector to carry out two essential functions:

1. **Searching for Problems:** The problem service provides server communication APIs for fetching a list of problems based on a user query.
2. **Displaying Problems:** It also provides server communication APIs for fetching all details of a particular user selected problem.

7.3.3.2 Auxiliary Service

All non-essential functionalities of the PRIORITY frontend are exposed through the auxiliary service. The current implementation only supports creation of Angular Material Snackbar component. This is used to display useful messages to the user in a snackbar, a kind of message box, which expires after a predefined time limit i.e., the message box component is auto-destroyed after a preset duration.

7.3.3.3 Problem Search Component

This component provides one of the most important views in the PRIORITY frontend. It houses application logic for interacting with the user by allowing selection and clearing of problem associated labels. It also contains event handler APIs to dynamically display a tabular view of clickable

problem titles, populated based on a user query. The user can further click on any of these problem titles which then navigates to a different route to display the problem details. With the help of the imported label selection component, it displays a selection of labels and meta-data for the user to choose from. After selecting appropriate labels and tags, the user can then make a query to the backend by click of a button as shown in Figure 7.8. The component then dynamically populates a table with problem titles, fetched from the backend server through REST API calls.

Alongside this, the component also provides minor capabilities like 'scroll to top', dialogs to display useful information to the user etc. for making the interface more user friendly.

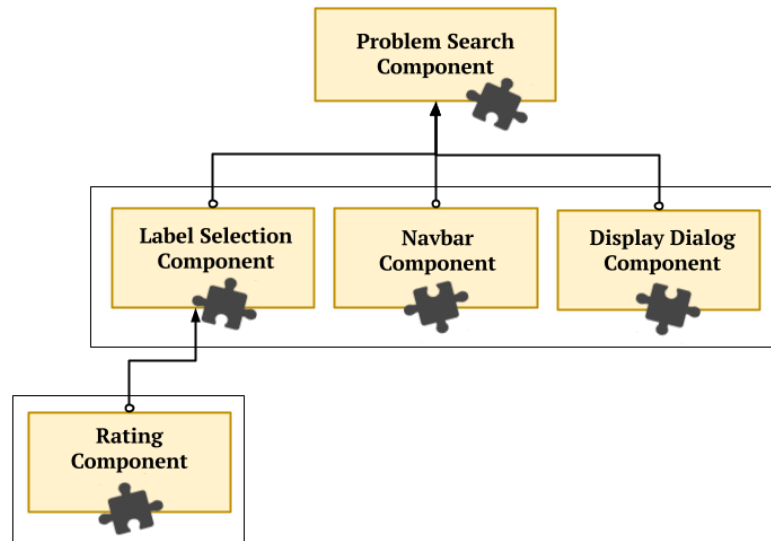


Figure 7.7: Problem Search Component

No.	Title
1	SEM2-15-16-Substring
2	Who is older?
3	SEM2-17-18-03_Loops_Say the Words
4	L1S1P2:The Imitation Game
5	Partial Palindrome
6	Partial Palindrome

Figure 7.8: Problem Search Component View

7.3.3.4 Problem Display Component

This component provides another very important view in the PRIORITY frontend. The problem display component view makes an aesthetic presentation of various problem details by using expansion panels. Only those expansion panels are initially in expanded state to that show information which is most relevant to the user. To view the other details about the problem the user can click to expand on those panels, which are hidden by default. The component houses application logic for interacting with the user by allowing the user to view/ hide various problem details and copy the details by click of a button from appropriate sections.

Alongside the problem display functionality, the problem display component also provides the capability to collect feedback from the user. It offers two ways in which users could submit feedback: With the help of an included rating component, the component view allows users to submit a star-rating by clicking on the feedback bubble.

The component also incorporates a label selection component, meant to collect user feedback on the problem details, that sit flush at the bottom of the view, and provides the necessary method APIs to submit the user feedback to the backend.

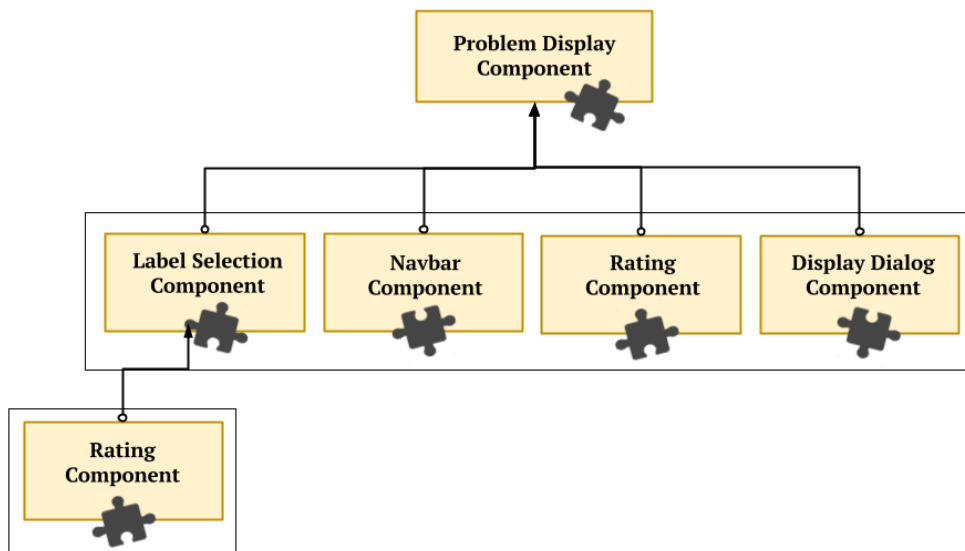


Figure 7.9: Problem Display Component

7.3.4 Feedback Module

The biggest challenge that PRIORITY faces currently is, to learn from data. PRIORITY, which is essentially a search engine of ESC101 and which is used as an information retrieval system or a recommendation system, must learn to make good recommendations for the users of the portal. Whenever the user makes a query, the system must determine the most relevant set of problems to display corresponding to a specific query made by the user.

Given that much of the available problems from the dataset are labelled automatically using the AI-system, the collection of labels right now might not be fully accurate due to inappropriate tagging. The list of labels may even be incomplete, as in there could definitely be problems that missed out on some labels. Some of them will be relevant, some of them will be mistakes, they may

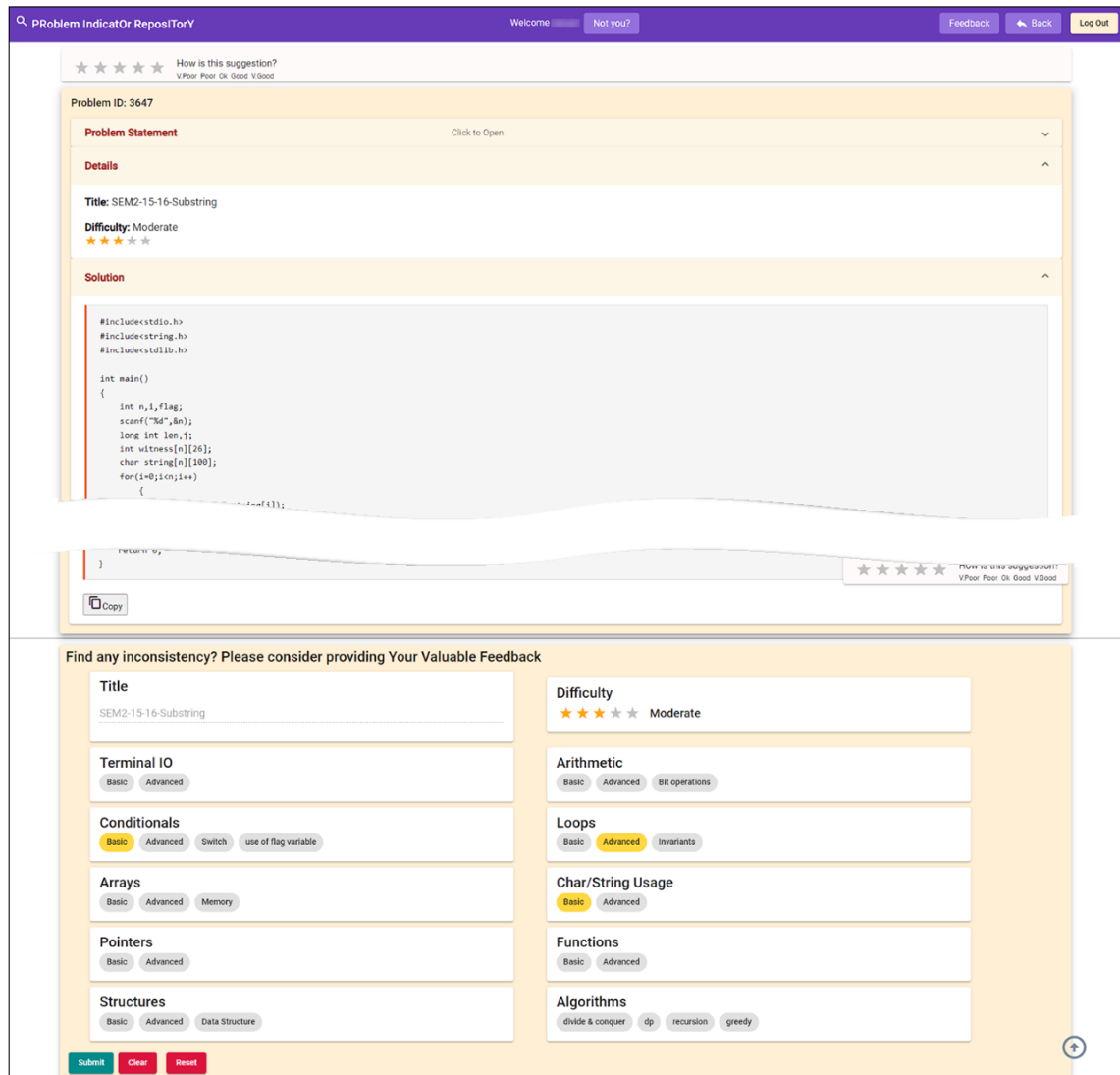


Figure 7.10: Problem Display Split Component View

be inappropriate. Given this, it is imperative that the system improves its performance as it gets to understand the data more and more. One of the ways of doing this is to study the impressions that it leaves on the users, as it makes its recommendations for user queries. The quickest and the easiest way to do this is by collecting feedback from the users. This feedback corroborates the AI-system and helps enhance its performance.

This feedback feature module enables collecting all kinds of feedback from users of the application. It facilitates collection of feedback from the portal users. The set of components and services it encapsulates enable these different ways to collect feedback. The feedback is collected by the system in various forms, which can be broadly classified into two types:

- **Explicit Feedback:** This is the kind of feedback where in we make specific provision within the application where users of the application could submit a feedback, based on the experience they have had using it. In the next section we describe a few ways in which the portal makes provision for the users to submit feedback.
- **Implicit Feedback:** Not every user would be gracious and patient enough to submit such

explicit feedback and always. So there has to be a way to get feedback implicitly as the users navigate through the portal. The portal achieves this by tracking the users albeit only aspects related to the usage of the portal itself. User tracking is quite popular, especially among social media platforms like Facebook, YouTube and the likes. It is of pivotal importance to search engines like Google and Bing too. They do this by tracking user cookies and using all other tracking platforms, which hold a wealth of information about the user. PRIORITY doesn't track cookies or intercept any other data about the user, other than just the portal usage statistics.

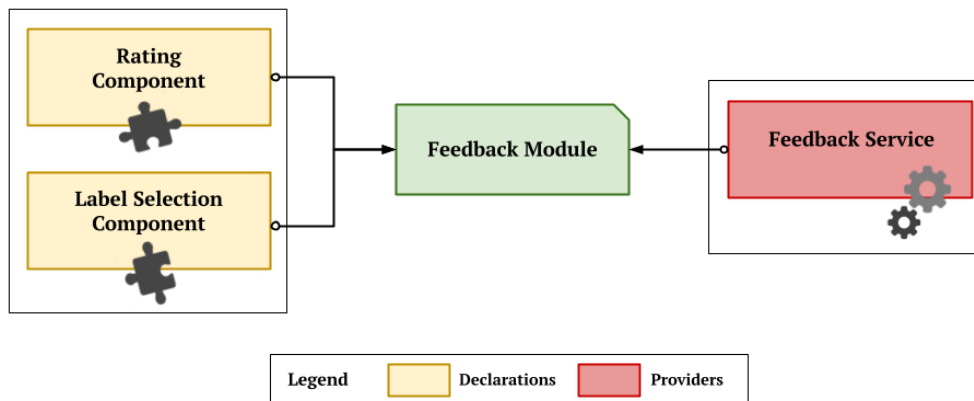


Figure 7.11: Feedback Module

7.3.4.1 Feedback Service

This is the most important part of the Feedback Module. The core functionality of the feedback service is to save and submit the user feedback. It provides the necessary method APIs for communicating with the backend APIs to store feedbacks appropriately. There are three ways of submitting an explicit feedback by the user:

- **Star-rating:** The user can submit a star-rating based on his experience using the portal.
- **Problem Feedback:** The user can also suggest better tagging of the problems by submitting a fresh set of label annotation pertaining to a particular problem.
- **Textual Feedback:** The user can submit any other comment they like to, regarding the portal features or their experience using it.

Apart from collecting explicit feedback this way, the feedback service also enables collection of implicit feedback from the users. While not all users would be gracious enough to give us a rating, or submit a comment, or better labels for a problem, there may be some who simply click on a 5 or a 1, something like that, for everything that they see. This is analogous to people compulsively clicking on the like button on Facebook on every post that they see. While such feedback definitely a valuable source of information, we can't simply rely on this particular feedback for finding out if the recommendations were good or not, rather it has to be used in conjunction with all other feedback mechanisms in place.

As noted earlier, not every user would submit such explicit feedback and always, so there it also aggregates feedback implicitly as the users navigate through the portal. The portal achieves this by tracking the users albeit only aspects related to the usage of the portal itself. PRIORITY doesn't track cookies or intercept any other data about the user, other than just the portal usage statistics.

Another trick that search engines use to figure out whether their recommendations were good or not and that is 'dwell time'. If a user clicks on a page on Google or Bing, they will go to the page, and simultaneously Google starts a counter. It will check how much time the user spends before they come back to the search page, before they come back to google.com. If they spend a very short time on the page, Google gets a signal that it may not have been a very good recommendation. If it had been very good, the user wouldn't necessarily have come back to google.com so quickly. Consider another scenario where the user performs a search on these search engines, suppose they click on the first query and they come back, then they click on the second query and then they don't come back or else you come back after a long time, this behavior provides a hint to Google that the first ranked query was not a good one, and that the second ranked query was much better, or maybe not the best, but much better than the first one. So, the next time the user makes the query, Google might flip the order. So, it will show the second rank query first and the first rank query second or maybe something else. This is an implicit way of gaining insight into what did the user think about the recommendations that the search engine made.

With the help of the feedback service, PRIORITY does exactly that, it keeps account of the dwell times of different users i.e., the amount of time they spend on a particular problems, as they make use of the portal. To support this, the portal records the rank of the displayed problem in the suggestions list, along with the set of the labels that were active for the particular user query. Aside this, it also tracks certain events that are triggered by user actions on the portal. For instance, the portal records click events, such as when a user clicks to expand the problem description (which is collapsed by default in the problem display view), when the user clicks to copy the problem etc. Specifically the portal keeps track of the following events and records time information on these events as well:

- **Navigation:** Whenever a user navigates to or away from the problem display page, it records the event.
- **Copy:** Whenever a user copies a piece of code (or problem detail), by clicking on the provided Copy button, it records the event.
- **Login/ Logout:** Whenever a user logs in to the portal or logs out from it, it records the event.

7.3.4.2 Rating Component

The rating component contributes a 5 star feedback rating bubble to its view. It provides a star-rating system which allows to record the users' experiences with the various problem recommendations of the system. The star-rating is a like-dislike system implemented by PRIORITY. At the top of the problem display page, the user sees a feedback bubble, that is in place for collecting a star rating from the users of the portal. This feature allows users to submit a star-rating on a scale

of 5 stars. This helps gauge the usefulness of the specific problem displayed to the user (and the rank of its display in the search results). A rating of 5 would mean that the user was very happy with the suggestion, which is something that they were exactly looking for. A rating of 1 would mean that the suggestion was way off and totally inappropriate rather a mistake made by the system. This is very analogous to the Facebook like-dislike feature, only it has 5 levels of feedback in place of two.

Along with the rating, it also records the rank of the problem in the table of results displayed on the previous page, along with the user query (the set of selected labels).

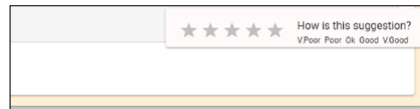


Figure 7.12: Rating Component View

7.3.4.3 Label Selection Component

The label selection component is central to the feedback collection mechanism of PRIORITY. It contributes to displaying all of the labels and different meta-data tags related to problems as shown in Figure 7.14. This component serves two main purposes:

1. On the problem search page, the component view lists all the labels and meta-data, all de-selected by default, with which users can fire a search query. It is used to record the user query i.e., keeps track of the set of labels selected by the user, emits an event to notify the application for querying the backend, when the user clicks on the submit button.
2. On the problem display page, the component view lists all the labels and meta-data of problems, keeping all those corresponding to the particular problem that is displayed, selected by default. The users can modify these selections and subsequently submit it as an explicit feedback on the problem labels. This way, it records a fresh set of label annotation from the user corresponding to the problem displayed. It does so with help from the feedback service to communicate with the backend for storing the feedback.

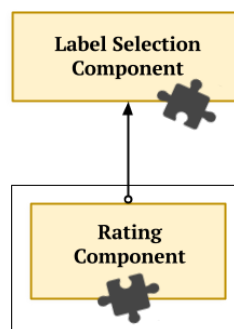


Figure 7.13: Label Selection Component

Select labels to search

Difficulty
★★★★★ Moderate

Arithmetic
Basic Advanced Bit operations

Loops
Basic Advanced Invariants

Char/String Usage
Basic Advanced

Functions
Basic Advanced

Algorithms
divide & conquer dp recursion greedy

Terminal IO
Basic Advanced

Conditionals
Basic Advanced Switch use of flag variable

Arrays
Basic Advanced Memory

Pointers
Basic Advanced

Structures
Basic Advanced Data Structure

Share your Thoughts!
Submit

Search Clear

Figure 7.14: Label Selection Component View

7.4 Application Flow

In this section we describe a typical way in which any user would use the portal. Figure 7.15 summarises the interactions of the user with the PRIORITY frontend.



Figure 7.15: PRIORITY Portal Application Flow

1. The user would first need to log in to the portal through a web browser. For this they already need to be a recognized user for whom a user account has been created and credentials shared earlier. If not, they have a provision to request for signing them up by writing to the PRIORITY team (this info is displayed upon clicking the Sign Up button on the login page).
2. After logging in, the user would land on the problem search page, where they would see a set of labels/ tags, all de-selected initially. The user can then choose the relevant set of labels and make a query to the backend server by clicking on the search button on that page. The server would then return a filtered set of problems after having matched them against the labels selected in the backend. The user will view this as a table of clickable problem

titles.

3. The user would then click on any problem title of their interest, this would navigate them to the problem display page, which is populated with all the details of the specific problem chosen. The bottom of the page contains all the labels as seen in the previous page, except now all the labels which are associated with the problem will be selected by default (which could be very well different from the exact set of labels selected by the user previously). The user would then review the problem and could also leave behind a feedback on the relevance of the problem to them through a star-rating or even submit a feedback suggesting corrections to the label selections shown on this page.
4. The user could then navigate back to the login page or just logout of the portal (the user could logout of the portal any time or from any where). If they come back to the problem search page, the persistent results from the previous search will still be visible to them and they can choose to explore more problems or simply modify the selection of labels and fire a query again to refresh the list. The cycle of events then repeats from step 1, as depicted in Figure 7.15.

Back-End Server

Contents

8.1 Database Description	65
8.1.1 Conceptual Framework	65
8.1.2 ER Model	66
8.2 System Overview	70
8.2.1 Architectural Overview	70
8.2.2 Development View	72
8.2.2.1 Web Service	72
8.2.2.2 Middleware	72
8.2.2.3 Database	73
8.3 API Description	74

Abstract *In this chapter we will go over some of the details of the backend server that is coupled with the frontend portal described in chapter 7. We begin the chapter by giving a conceptual framework of the system. We then give a brief description of the relation between various entities in the system and the database schema associated with their storage. Then we look at an overview of the system architecture. This is followed by a section which highlights various aspects of development of the backend. The chapter concludes by documenting the various tables in the database system and descriptions of the APIs that are exposed by the backend.*

8.1 Database Description

As will be explained elaborately in section 8.2, the PRIORITY backend mainly comprises of a Django application that entertains client requests through RESTful APIs, interacts with the backend database and sends responses appropriately. The database plays an important role in any backend system. In the following sections we describe some of the entities stored in the database and the terms related to them, and then draw up a relation between these different entities.

8.1.1 Conceptual Framework

In this section we describe the various entities of the system and some terms related to them.

1. **Backend:** This refers to the backend sub-system of PRIORITY.
2. **User:** This refers to an entity who has been authorized to access the PRIORITY system.

3. **Problem:** This refers to an instance of a programming question, consisting of the problem statement, the gold-solution to it, a set of labels that pertain to it among other things.
4. **Feedback:** This refers to an instance of a list of all meta-data, including labels, describing a problem, which is submitted by a User.
5. **Feedback Rating:** This refers to an instance of a 5-star rating, submitted by a user.
6. **Actions:** Actions are means by which the user provides input to the application. For PRIORITY, it mainly refers to the following user events:
 - When the user requests the backend for a list of problems relevant to their query.
 - When a user requests the backend for details of a particular problem
 - When the user submits a feedback
7. **Click:** This refers to an instance of the event when a user selects a particular problem or copies description related to the problem.
8. **Text Feedback:** This refers to an instance of textual comment(s) submitted by the user.

8.1.2 ER Model

The entities, as described in subsection 8.1.1 refer to the humans interacting with the PRIORITY portal, the problems, as well as the information that they generate in due course of their interactions.

Figure 8.1 depicts the various entities and how they are related with each other. The User forms an external entity to the PRIORITY system. The Problem forms the main internal entity. All of the other entities are generated as a result of the User interactions

- A Feedback is generated as a result of User interacting with a Problem.
- A Textual Feedback is also generated as a result of User interacting with the System.
- A click is generated as a result of User interacting with the System or a Problem.
- A Feedback Rating is generated as a result of User interacting with the System and a Problem.

What follows is a tabular representation of the database schema, that describes the database tables associated with each of the prime entities discussed thus far.

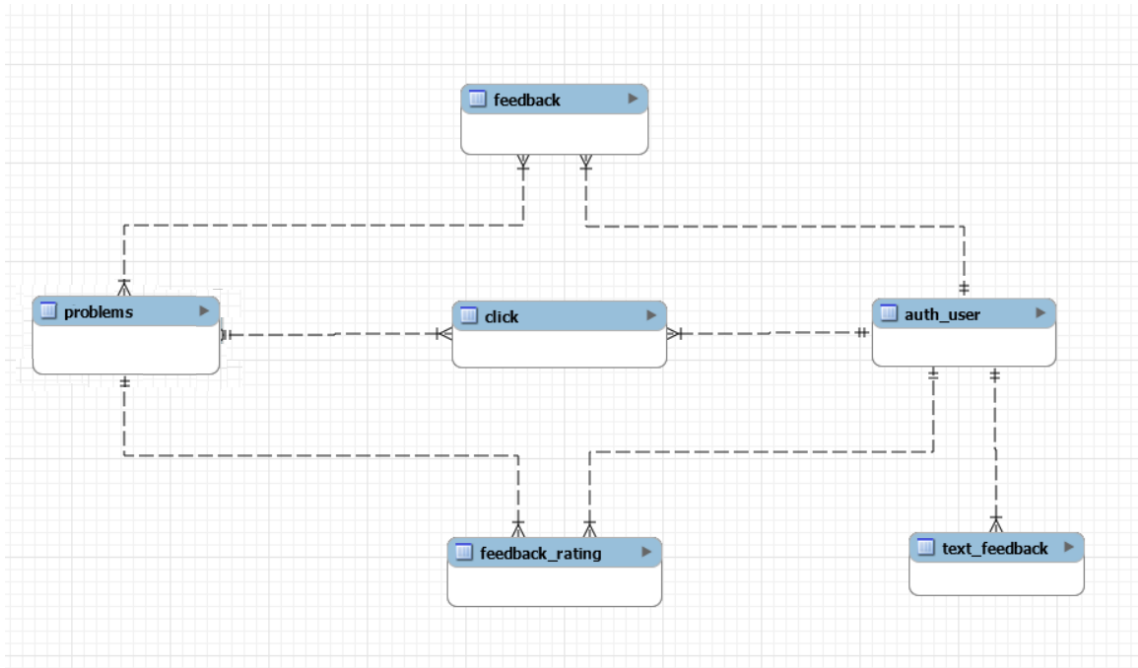


Figure 8.1: Entity Relationship Diagram

CLICK		
Field Name	Data type	Description
id	int	primary key
start_time	datetime	
end_time	datetime	
position_of_recomendation	int	
query	varchar	
problem_id	int	References Problem table
auth_user_id	int	References USER table
category	varchar	

Table 8.1: Click Table

USER		
Field Name	Data type	Description
id	int	primary key
password	varchar	
last_login	datetime	
is_superuser	tinyint	
username	varchar	
first_name	varchar	
last_name	varchar	
email	varchar	unique
is_staff	tinyint	
is_active	tinyint	
date_joined	datetime	

Table 8.2: User Table

FEEDBACK		
Field Name	Data type	Description
id	int	primary key
difficulty	int	
terminalio_basic	int	
terminalio_advanced	int	
arithmetic_basic	int	
arithmetic_advanced	int	
arithmetic_bit	int	
conditionals_basic	int	
conditionals_advanced	int	
conditionals_switch	int	
conditionals_flag	int	
loops_basic	int	
loops_advanced	int	
loops_invariants	int	
arrays_basic	int	
arrays_advanced	int	
arrays_memory	int	
pointers_basic	int	
pointers_advanced	int	
charstring_basic	int	
charstring_advanced	int	
functions_basic	int	
functions_advanced	int	
structures_basic	int	
structures_advanced	int	
structures_ds	int	
algorithms_dc	int	
algorithms_dp	int	
algorithms_recursion	int	
algorithms_greedy	int	
problem_id	int	
auth_user_id	int	
title	varchar	

Table 8.3: Feedback Table

PROBLEM		
Field Name	Data type	Description
ID	int	Primary key
title	varchar	
statement	longtext	
solution	longtext	
template	longtext	
difficulty	int	
is_labeled	int	
terminalio_basic	int	
terminalio_advanced	int	
arithmetic_basic	int	
arithmetic_advanced	int	
arithmetic_bit	int	
conditionals_basic	int	
conditionals_advanced	int	
conditionals_switch	int	
conditionals_flag	int	
loops_basic	int	
loops_advanced	int	
loops_invariants	int	
arrays_basic	int	
arrays_advanced	int	
arrays_memory	int	
pointers_basic	int	
pointers_advanced	int	
charstring_basic	int	
charstring_advanced	int	
functions_basic	int	
functions_advanced	int	
structures_basic	int	
structures_advanced	int	
structures_ds	int	
algorithms_dc	int	
algorithms_dp	int	
algorithms_recursion	int	
algorithms_greedy	int	

Table 8.4: Problems Table

TEXT FEEDBACK		
Field Name	Data type	Description
id	int	Primary key
auth_user_id	int	References auth_user table
query	varchar	
no_of_results	int	
version	varchar	
feedback	varchar	

Table 8.5: Text Feedback Table

FEEDBACK RATING		
Field Name	Data type	Description
id	int	Primary key
time	datetime	
position_of_recomendation	int	
query	varchar	
rating	int	
problem_id	int	References problem table
auth_user_id	int	References user table

Table 8.6: Feedback Rating Table

8.2 System Overview

8.2.1 Architectural Overview

PRIORITY backend was built using the Django-REST framework. The Django-REST framework serializes data from the Django ORM and allows access/ updates via a RESTful API. Django ORM creates and manages database models and queries. There are several advantages of Django and using the Django-REST framework, we note some of these in the following points.

- **Fast:** Django framework is not only fast in its operation, but also enables fast paced web development, by providing all of the common functionality that the users need to build and deploy web applications. Django-REST framework is no different, and makes building REST APIs a breeze.
- **Well Documented:** The extensive, up-to-date documentation of Django-REST framework and a good community support makes it an apt choice as a backend development framework.
- **Versatile:** It is a robust backend web development framework and provides various in-built functionality including user authentication, security modules, a huge array of utility libraries etc., right out of the box that makes it extremely useful.
- **Secure:** Any application hosted on the web is subject to common exploits like SQL Injection, XSS vulnerability etc., the Django-REST framework takes security seriously and helps developers avoid these common pitfalls.
- **Serializable:** One of the main reasons to use Django REST Framework is that it makes serialization very easy. Serializers act as translators between Django model instances and their representations such as JSON.
- **Scalable:** While providing all of the above advantages, Django-REST framework is very responsive and provides good scalability.

Django is based on the MTV (Model-Template-View) architecture, a design pattern for web applications. In contrast to the common MVC architecture, Django views are called templates and controllers are generally called views.

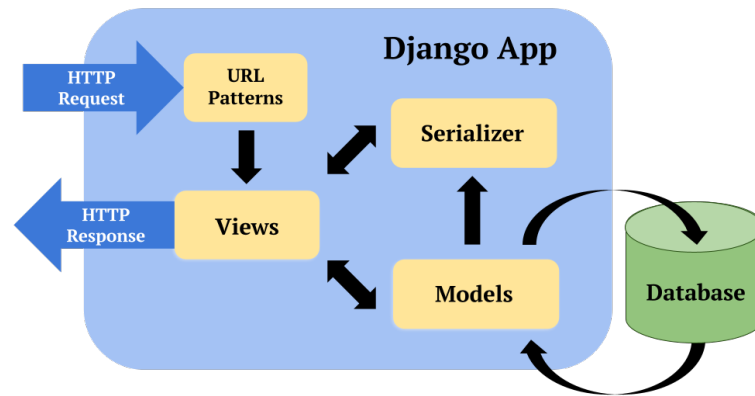


Figure 8.2: Django App Architecture

- **Model:** Django models are an interface to data. They simplify various tasks of a Structured Query Language, by organizing tables into models. Generally each model maps to a single database table.
- **Template:** Template comprises of static HTML content and some syntaxes pertaining to how the dynamic content will be structured. Django generates dynamic HTML web pages based on templates.
- **View:** View is the user interface. It is typically consists of HTML, CSS, JavaScript files rendered in the browser. A view, in Django, is essentially a python function that takes a web request as input and returns a web response as output.

PRIORITY backend, essentially a Django application built using the Django-REST framework, follows a similar design paradigm. Figure 8.3 provides a good illustration of the architecture of the PRIORITY backend.

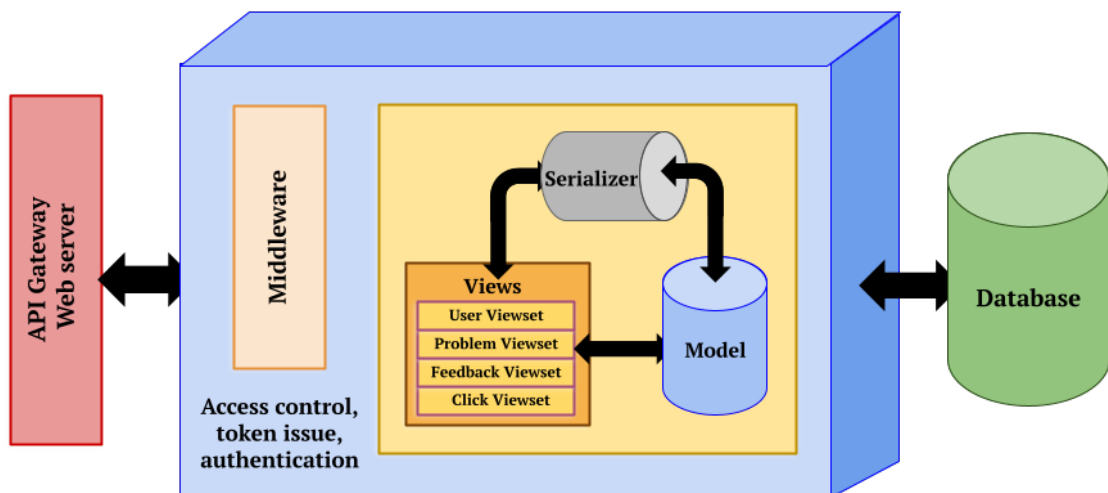


Figure 8.3: Architectural Overview of PRIORITY Backend

PRIORITY backend is run as a hosted Django web server, that exposes the API end-points developed with the Django-REST framework. All of the middleware components that the PRIORITY backend uses is imported from the in-built functionality provided by Django. The backend

database used is MySQL, for persistent storage of all application related data. The backend runs as one service that responds to all of the different requests made by the client. In our case, the Angular application sits in the front-end and is what forwards all requests corresponding to user actions to the backend.

8.2.2 Development View

8.2.2.1 Web Service

- **PRIORITY** backend is built using the MVT (or MTV) architecture. The service implemented at the backend follow this design paradigm. The `views.py` file essentially fulfills the role of controllers
- The API routes are specified in the designated `urls.py` file. The global configuration pertaining to the Django app is all specified in the default `settings.py` file.
- Models are a logical definition of the actual database tables and provide a nice interface to access the data. All of the different models are placed in the `models.py` file.
- Serializers enable conversion of complex data such as querysets and model instances to native Python datatypes, so that they can be easily rendered into JSON, XML or other content types. Serializers also provide deserialization, allowing parsed data to be converted back into complex types. Serializers which are used for both reading and writing data, can also be used for field validations. All the serializers are defined in a dedicated `serializers.py` file in the Django app.

8.2.2.2 Middleware

Middleware is a framework of plugins into Django's request/response processing. Each middleware component is capable of providing a specific functionality. **PRIORITY** backend borrows almost all of the different middleware functionality from Django. We make a brief note on the important ones in the following points, for a more detailed description we refer the reader to the official documentation¹.

- **SecurityMiddleware**: This component provides many security enhancements to the request/ response cycle of the Django app. Currently the backend service is run on http and doesn't make much use of the capabilities provided by this component.
- **CommonMiddleware**: This is more of a convenience component. It provides functionality such as appending 'slash', prepending 'www' in URLs etc.
- **CsrfViewMiddleware**: This component gives protection against Cross Site Request Forgeries by adding hidden form fields to POST forms and checking requests for the correct value.
- **SessionMiddleware**: This component enables session support. This is used by the `AuthenticationMiddleware`.

¹<https://docs.djangoproject.com/en/3.2/ref/middleware/>

- **AuthenticationMiddleware:** This component appends the user attribute, representing the currently-logged-in user, to every incoming HttpRequest object.
- **XFrameOptionsMiddleware:** This component provides protection against clickjacking, which is a type of attack that occurs when a malicious site tricks a user into clicking on a concealed element of another site.

8.2.2.3 Database

PRIORITY backend uses MySQL database for persistent storage. As a relational database, there are different tables to store data.

- **User Tables:** These are default Django generated tables that are used for authentication and maintaining user sessions, essentially managing user access to the backend.
- **Click Table:** This table is used for registering various tracking events from the user. The data collected forms implicit feedback extracted from the users, as they interact with the PRIORITY frontend. It contains the user id, query made by the user, the position of recommendation (of the problem link the user clicks on) among other things.
- **Feedback Tables:** There are a few tables in the backend database for registering explicit feedback submitted by the users of PRIORITY portal.
 1. **Feedback Table:** This table registers the fresh set of label annotation submitted by the user on any particular problem.
 2. **Feedback Rating Table:** This table registers the 5-star rating submitted by the user.
 3. **Text Feedback Table:** This table registers the textual feedback submitted by the user.
- **Problems Table:** This table contains all of the problems, along with their labels. There is a field in the table that indicates whether the problem was labelled manually or had labels generated by the machine learning algorithm in the backend.

More details on the database schema can be found in subsection 8.1.2. The PRIORITY backend also uses the Django ORM for mapping complex data obtained from the database to defined model classes. It also equipped with an SQL generator engine, that automatically and dynamically generates SQL queries.

8.3 API Description

In this section we go over all of the APIs exposed by the backend web server.

Backend API: Create User	
URL	/user
HTTP Method	POST
Action	The API takes 2 parameters username and password and creates a new user in the auth_user table with these credentials. The password is hashed before its saved.
Return Data	A JSON object consisting of the provided username and hashed password alongwith the id of the new user just created.

Backend API: Authenticate User	
URL	/auth
HTTP Method	POST
Action	The API takes 2 parameters username and password and checks if the user with these credentials exists in the database.
Return Data	If the user exists then it returns the user id as well as a authorization token that needs to be re-sent to the server in all subsequent requests. Otherwise it returns an error message stating "Unable to login with given credentials"

Backend API: Text Feedback	
URL	/TextFeedback
HTTP Method	POST
Action	The API takes 5 arguments and stores a textual feedback from the user into the text_feedback table. The arguments are as follows: <ol style="list-style-type: none"> 1. user : The user-id. 2. query : The query (set of tags) the user last searched for. 3. no_of_results: The number of results returned in last query. 4. version : The version of PRIORITY being used. 5. feedback : The feedback that the user provided.
Return Data	Returns the same data user entered, along with the auto generated id.

Backend API: Feedback Rating	
URL	/FeedbackRating
HTTP Method	POST
Action	<p>The API takes 6 arguments and stores the rating for a suggestion from the user. The arguments are as follows:</p> <ol style="list-style-type: none"> 1. user : The current user-id. 2. problem : The problem-id of the problem for which the user is providing feedback. 3. query : The query (set of tags) the user last searched for. 4. time : The system time at which the rating was provided. 5. position_of_recommendation : The rank (in the list of results) at which this problem was suggested to the user. 6. rating : Number of stars 1-5 given by the user.
Return Data	Returns the same data user entered, along with the auto generated id.

Backend API: Feedback	
URL	/Feedback
HTTP Method	POST
Action	<p>The API takes feedback from the user in the form of a fresh set of labels annotation provided by the user for a given problem and stores it in the feedback table. The arguments include the user-id and the problem-id, along with labels annotation. The user can also provide a difficulty score.</p>
Return Data	Returns the same data user entered, along with the auto generated id.

Backend API: Passive Feedback	
URL	/click
HTTP Method	POST
Action	<p>The API takes 7 arguments and stores the rating for a suggestion from the user. The arguments are as follows:</p> <ol style="list-style-type: none"> 1. user : The current user-id. 2. problem : The problem-id of the problem for which the user is providing feedback. 3. query : The query (set of tags) the user last searched for. 4. start_time : The system time at which the user entered the ProblemDetails page. 5. end_time : The system time at which the user left the ProblemDetails page. 6. position_of_recommendation : The rank (in the list of results) at which this problem was suggested to the user. 7. category : The type of passive feedback.
Return Data	Returns the same data user entered, along with the auto generated id.

Backend API: Problem Details	
URL	/Problems/\$problemid
HTTP Method	GET
Action	None
Return Data	Returns a JSON object with details of the problem specified by the problem-id parameter.

Backend API: Problem Search	
URL	/ProblemSearch
HTTP Method	POST
Action	<p>The API takes a list of the labels as parameters and uses them to search the Problems table to find most relevant problems which match the user selected labels. The search is performed at 2 levels. First we try to find all problems whose set of labels is a super-set of the user selected labels. We rank these problems by comparing how many labels match between the label set of the problem and the user selection i.e. if the set of labels for the problem is exactly the same as the user selection it would be ranked highest. If there is a tie then the labels are sorted based on their complexity. We determine the complexity of a label based on how late in the programming course is the topic corresponding to the label taught (assuming topics covered later are more advanced compared to the ones taught earlier). For example, complexity of functions is more than that of loops or conditional statements. Now, if there are no such problems having labels set which is a super-set of user selection then we start dropping the labels selected by the user one at a time and keep searching for relevant problems that match the new set of labels. We drop the labels in reverse order of the labels' complexity.</p>
Return Data	Returns a array of JSON objects where each JSON object has details of a problem.

Deployment and User Feedback

Contents

9.1	Deployment	79
9.2	User Feedback	80
9.2.1	Background	80
9.2.2	Survey Summary	81
9.2.3	Survey Statistics	82

This chapter describes, in brief, the deployment of PRIORITY and the results of a survey based on its usage in a live environment where it was deployed. The PRIORITY portal is hosted on the Indian Institute of Technology Kanpur, CSE department cloud infrastructure and hence is accessible only over the intranet for now. Since we do not have a public domain, we use a self signed certificate to serve PRIORITY over https. It is currently used by the ESC101 tutors and instructors only. Thus the number of users would be anywhere between 12-20, which is relatively a small user base. Owing to this fact it was not required to configure a load balancer or any in-memory caching.

9.1 Deployment

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow us to pack up an application with all of the parts and deploy it as one package. PRIORITY is deployed in 3 different Containers viz., one for the Angular front-end, one for the REST APIs and one for the Database as shown in Figure 9.1. Each container is more or less independent of one another. Compose is a tool for defining and running multi-container Docker applications. We use Docker Compose to make our deployment easier.

1. **Front-end Container:** As discussed in previous sections we use Angular to develop the front-end for PRIORITY. NGINX is a free, open-source, high-performance HTTP server and is often used to deploy Angular applications. We use NGINX to serve our web-app. The front-end container consists of all of the static HTML, CSS, JS files as well as the Angular code. The app is served over both http (port 80) and https (port 443).
2. **Back-end Container:** The back-end container contains the code for the RESTful APIs. It exposes the APIs at a specific port. The front-end communicates with the backend by sending HTTP requests on this port.
3. **Database Container:** As stated earlier, we use MySQL as a database for our application. The Database Container contains an image for MySQL service. When the container is

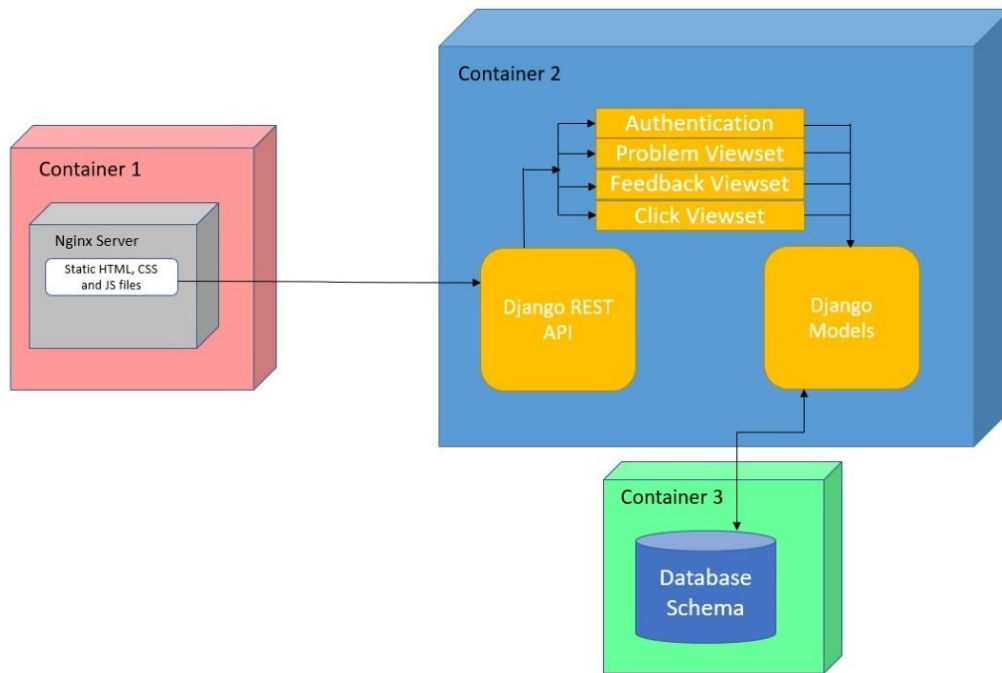


Figure 9.1: PRIORITY Deployment Architecture

created for the first time all the labelled problems are loaded into the Problems table. It is to be noted that this is a one-time activity. The Database Container interacts with the Back-end Container and serves all the necessary data.

9.2 User Feedback

9.2.1 Background

PRIORITY was made available to tutors of ESC101 course, in its previous offering the past semester. To recollect, one of the tutor's main responsibility is to set questions for the students who have undertaken the course. The questions were set in roughly two phases.

- **Phase I:** This phase spanned from week 1 to week 5 of the course. During this phase PRIORITY was enabled for use by the tutors.
- **Phase II:** This phase spanned from week 6 onwards to the end of the course. During this phase PRIORITY was disabled and not available for tutors to use.

At the end of Phase II, we asked the tutors about their engagement with PRIORITY and experience using it by conducting a survey. The survey included the following aspects

1. Number of questions set in Phase I as well as Phase II
2. Reliance on PRIORITY for helping them set questions in Phase I
3. Feedback on the ranked list of problems based on user query
4. The time and effort saved in setting questions with PRIORITY's help

5. Feedback on the features that PRIORITY provides

There were a total of 8 tutors who completed the survey, all of them students from fourth year of B-Tech/ Dual Degree program. All of them were at least moderately proficient with the C programming language, a couple of them extremely proficient. The bar graph in Figure 9.2 shows the distribution of the number of tutors plotted against the number of questions they created in both Phase I and Phase II.

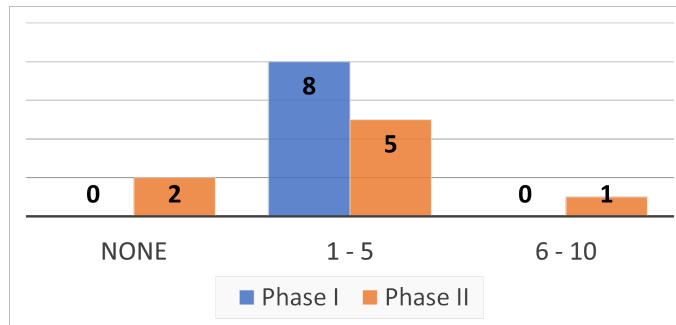


Figure 9.2: Plot of # of tutors (vertical) against range of # questions created (horizontal)

9.2.2 Survey Summary

What follows is a summary of the survey statistics. This section also highlights the key take-aways from the survey.

- **Usage:** 5 out of the 8 tutors made extensive use of PRIORITY. A couple of them believed that they would lose innovation if they used PRIORITY, reasons out their limited interaction with the portal.
- **Experience:** 6 out of the 8 tutors got relevant results in their first 5 queries, and in top 5 results among that, on an average. More than half of them reported that using PRIORITY helped them significantly reduce the time and effort involved in setting questions.

We also received some additional feedback we received requesting enhancements to features that PRIORITY already provides or some additional features. Some of them are listed below.

- **Natural language search:** We saw a demand for keyword-based natural language search feature.
- **Negative tags search:** This was some suggestion for forcefully excluding certain tags during search.
- **Displaying tags in problem search page:** This request probably came from being able to quickly modify queries. Currently all tags are only shown in the problem details page.
- **Better tagging:** Owing to the nature of the topics involved in some of the programming questions to be crafted, a couple of them reported that they had to play around with label combinations a bit to find suitable problems.

Overall, we received a very positive feedback about PRIORITY and its usefulness for ESC101, which is very encouraging to its continued development.

9.2.3 Survey Statistics

In this section, we will visualize some of the responses given by the tutors for the most important questions asked in the survey.

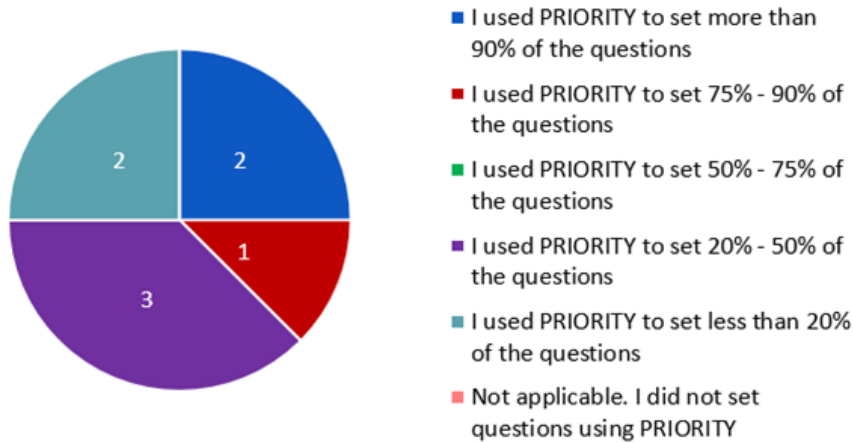


Figure 9.3: Fraction of questions set with assistance from PRIORITY (Phase I)

What we see in Figure 9.3 is perhaps one of the most important stats from the survey. The chart shows that about a third of all the tutors (shown in blue and red) used PRIORITY to set more than 75% of the questions. Another third of them (the one in purple) used PRIORITY to set up to 50% of the questions. A couple of them used PRIORITY sparingly.

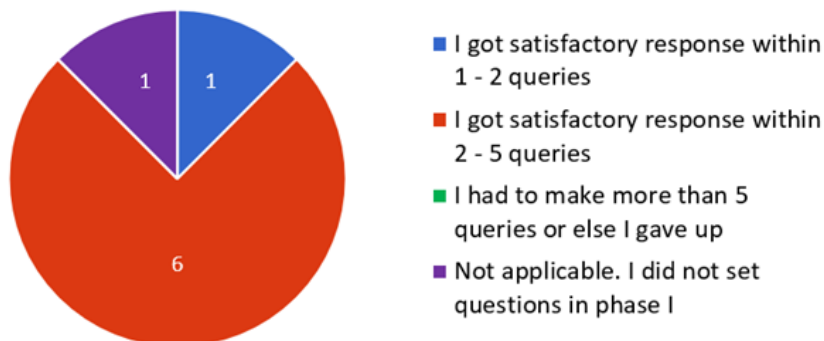


Figure 9.4: Average number of queries made before getting a 'satisfactory' result

Looking at Figure 9.4, we find that more than $\frac{3}{4}$ th of the tutors (shown in blue and red) found what they were looking for in less than 5 queries. Shifting our focus to Figure 9.5, we observe that a vast majority of the tutors (the ones in blue and red) also found satisfactory results in the top 5 results of their queries.

What we see on Figure 9.6 is perhaps the second most important stats from the survey. It shows that about $\frac{2}{3}$ rd of the tutors (shown in blue and red) found the tool to significantly or moderately reduce their time and effort when compared to setting questions in Phase II. Figure 9.7 indicates that all the tutors feel that a tool like PRIORITY can help reduce a good amount of time and effort required to set questions in courses such as ESC101.



Figure 9.5: Average ranked position of a 'satisfactory' result

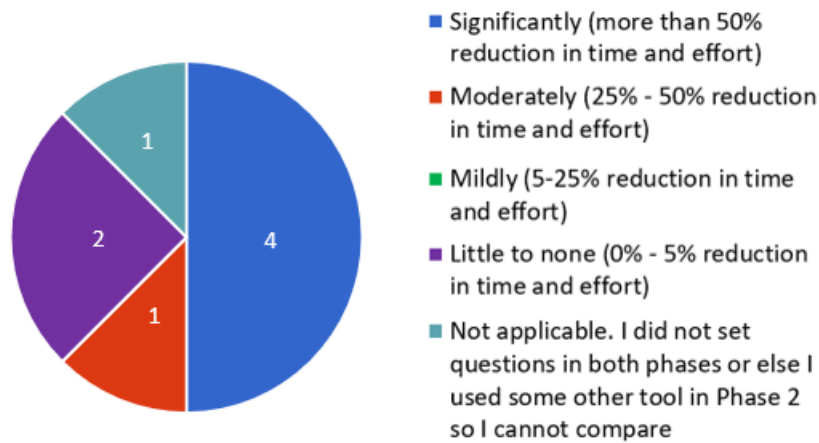


Figure 9.6: Reduction in time and effort to frame questions with help of PRIORITY as compared to without using PRIORITY

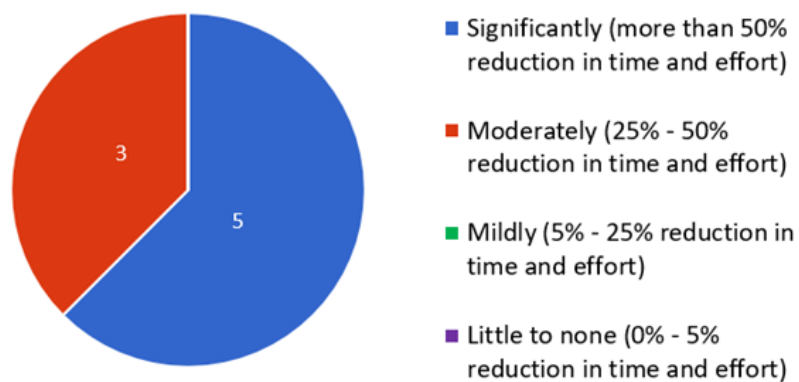


Figure 9.7: Estimation on reduction in time and effort to frame questions with the help of tools like PRIORITY

Conclusion and Future Work

In this part of the thesis we took a look at PRIORITY, a web application portal that provides users access to a search-able labelled inventory of computer programs in the C programming language from the past offerings of ESC101, an introductory programming course offered to freshman undergraduate students at Indian Institute of Technology Kanpur. Based on the results of a survey conducted on its usage in a live deployment environment, we received a very positive feedback about PRIORITY and its usefulness for ESC101. This is indeed encouraging to invest time and efforts in its further development.

The portal developed is a first generation implementation but very much usable. There is much scope for improvement in terms of features that could be added on top of the existing portal.

On the usability front, a couple of features that have been thought of and is likely to be seen in a future iteration of the web-app are - the ability to bookmark questions of interest and be able to share them with peers; access to an admin portal to visualize usage stats etc.

On the user management front, an important feature that might make the portal more user-friendly would be to enable a forgot password option. We could also provide users with an option to sign/ fill up some details for requesting access to the portal (which could be approved by moderation).

On the deployment front, the web portal currently runs on the intranet of Indian Institute of Technology Kanpur. To make the application more secure, to enable fool-proof SSL service, it could be hosted on a public domain and thereby obtain an authentic certificate from an authorized CA. Also currently the portal limits its usage to only tutors and instructors of ESC101. In the event that it opens up for the general student body, or for instance instructors from other institutions (once it goes public), then additional services such as load balancing etc. might need to be configured.

Bibliography

- [1] Umair Z. Ahmed et al. Compilation error repair: for the student programs, from the student programs. *International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, 2018.
- [2] Umair Z. Ahmed et al. Targeted example generation for compilation errors. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 327–338, 2019.
- [3] Brett A. Becker, Kyle Goslin, and Graham Glanville. The effects of enhanced compiler error messages on a syntax error debugging test. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education, SIGCSE '18*, page 640–645, New York, NY, USA, 2018. Association for Computing Machinery.
- [4] Lijia Chen, Pingping Chen, and Zhijian Lin. Artificial intelligence in education: A review. *IEEE Access*, 8:75264–75278, 2020.
- [5] Darshak Chhatbar, Umair Z. Ahmed, and Purushottam Kar. MACER: A modular framework for accelerated compilation error repair. *CoRR*, abs/2005.14015, 2020.
- [6] Rajdeep Das et al. Prutor: A system for tutoring CS1 and collecting student programs for analysis. *CoRR*, abs/1608.03828, 2016.
- [7] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. All syntax errors are not equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE '12*, New York, NY, USA, 2012. Association for Computing Machinery.
- [8] Björn Hartmann et al. What would other programmers do: Suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '10*, page 1019–1028, New York, NY, USA, 2010. Association for Computing Machinery.
- [9] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. Compiler error messages: What can help novices? In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, SIGCSE '08*, page 168–172, 2008.
- [10] Raymond S. Pettit, John Homer, and Roger Gee. Do enhanced compiler error messages help students? results inconclusive. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, SIGCSE '17*, page 465–470, New York, NY, USA, 2017. Association for Computing Machinery.
- [11] David Pritchard. Frequency distribution of error messages. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU 2015*, page 1–8, New York, NY, USA, 2015. Association for Computing Machinery.
- [12] Fahad Shaikh. *Advancements in AI-assisted compilation error repair and program retrieval*. M.tech. thesis, Indian Institute of Technology Kanpur, 2021.
- [13] V. Javier Traver. On compiler error messages: What they say and what they mean. *Advances in Human-Computer Interaction*, 2010, 06 2010.
- [14] Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. *Proceedings of the 37th International Conference on Machine Learning, Vienna, Austria, PMLR 119.*, 2020.

